Software Patterns

Software Patterns

SIGS Management Briefings present cutting-edge information on object-oriented topics. Written by experts in their respective areas, these clear and concise papers are the fastest way to get the latest findings by today's top OO professionals. For a full listing of SIGS Management Briefings, contact SIGS Books & Multimedia customer service department at (800)871-7447 (voice), (212)242-7574 (fax), http://www.sigs.com.

Software Patterns

James O. Coplien

B ell Laboratories, The Hillside Group

Originally published by:
SIGS Books & Multimedia
71 West 23rd Street, Third Floor
New York, New York 10010
http://www.sigs.com

Please send inquiries to cope@research.bell-labs.com.

SIGS Books ISBN 1-884842-50-X

Originally printed in the United States of America

Software Patterns

Software Patterns

Interest in patterns has fostered one of the fastest-growing communities in contemporary software design. Like many emergent software techniques in recent history—structured programming, data abstraction, and objects—they have captured the imagination of many in the industry. Most of these techniques fail to fulfill the expectations that propel them to such prominence, partly because immature ideas are exposed to the market prematurely, partly because of opportunism, and partly because a hopeful (or desperate) market yearns for the one key solution to all problems. This is a recipe not only for mismatched expectations but for misinformation, which in turn fuels expectations and so on.

I would like to claim that this Briefing is the final word on patterns: the arbiter of disagreements and splinter perspectives, the definitive word on how excited we should or should not be about patterns. It is not and cannot be such a work. The Hillside Group, one loosely affiliated group of designers who are actively pursuing pattern solutions to industry problems, is brought together by a vision which itself accommodates and celebrates a variety of individual visions. Dialogue continues on what is and is not a pattern, on what is and is not a good pattern; this work does not cap that dialogue but contributes to it. The pattern community values diversity: there are many things called "patterns" that share a small core of techniques, principles, and values, all of which have proven useful to programmers. There are, in fact, several pattern communities, some of them quite decoupled from each other. This paper attempts to capture the practice and foundations of an (admittedly ill-defined) mainstream pattern culture.

This work is an *apo1ogia*, both in the sense of "apology" and of a *defensio* or "defense" When we discuss patterns in public, we bear responsibility for setting reasonable expectations for patterns. Most new techniques are heralded by expectations that outstrip what the technique can deliver, and the danger for patterns to fail into this trap is immense. Patterns touch critical issues that are central to strategic software development, and they are important for that reason. But they are also just documentation, and rely on the insights of the people who create and use them. It is important to view patterns as one more tool in the designer's tool kit. Their success depends on people, and particularly on the most human aspects of software development and its culture. This paper accentuates that perspective: we want to accentuate the value of people in design, and to diminish hype.

Though this Briefing is the work of a single author, it reflects the work of dozens of individuals. It draws directly on the contributions of many pattern writers and pattern thinkers, which takes the material outside my immediate sphere of thought. In fact, the following pages probably bear more of my colleagues' words than my own; primarily, I provide the arrangement and the glue that tie the pieces together.

This is consistent with another important agenda of the pattern community: we would much rather read, use, and write patterns than talk or write *about* patterns. Premature abstraction is dangerous; it too easily distracts us from the ends to which we should be aspiring. So, although this briefing is an early attempt to abstract principles from the pattern community and explores the frontiers of its value system, it is grounded in "real stuff." Enjoy it as an exposition of patterns. Focus on the patterns, think about them, and understand them—then read the surrounding analyses and commentary.

I've let many commentators challenge and shape my own thoughts while writing this Briefing. Thanks to Bruce Anderson, Kent Beck, Stephen Berczuk, Frank Buschmann, Paul Chisholm, Ward Cunningham, Brandon Goldfedder, Janel Green, Richard Gabriel, Norm Kerth, Gerard Meszaros, Hans Rohnert, Doug Schmidt, and Bjarne Stroustrup for their comments, examples, and suggestions.

2 Software Patterns

A Word on Alexander

Patterns have their roots in urban design and building architecture in the work of Christopher Alexander, the indisputable inspiration for the software patterns groundswell. Most pattern publications pay him homage; sales of his architecture books have soared in the software community. The vocabulary of software patterns—*forces*, the term pattern itself, pattern *languages*—comes from Alexander.

However, most of the pattern community has let go of literal interpretations of Alexander. The pragmatics of software development seem to dictate iterative development; Alexander would have us apply patterns in monotonic, progressive order. All analogies break down somewhere. While early pattern pioneers looked for Alexanderian analogies under every rock, software pattern literature is taking increasingly bold departures from Alexanderian structures and forms. Experience with patterns has borne out ties to other sources, including earlier work in computer science on literate programming.

Alexander's greatest legacy to the pattern community is his vision and value system, but his vision is so foreign to most software practice that it is often lost in technical dimension of patterns. One goal of this briefing is to highlight the pattern value system so it is more broadly understood and appreciated.

Organization of the Briefing

The paper is organized to start with the central questions of patterns, moving on to more refined and advanced issues later on. The sections include:

1. ***What Is a Pattern?*** This section looks at the prevailing definitions of patterns, both historically and in contemporary software patterns. It also explains why patterns are important.
2. ***What Are Pattern Languages?*** Building on Section 1, this section defines collections of patterns called pattern Languages, the most important application of patterns to system design.
3. ***Pattern Domains.*** This section gives examples of software disciplines that are using patterns.
4. ***Classifying Patterns.*** With hundreds of patterns, how do designers find the ones they're looking for? This section investigates the organizing principles currently being used for software patterns.
5. ***Pattern Pragmatics.*** This section outlines the business benefits of patterns and how to introduce patterns into a development culture.
6. ***Generativity.*** Generativity is what distinguishes patterns from rules. It is a subtle but important component of the pattern culture.
7. ***The Pattern Value System.*** A value system has arisen within the pattern community that restores dignity to programmers. We also recognize several important legal and ethical issues.
8. ***History.*** How patterns found their way from Alexander's work into contemporary software architecture; the people and the events leading up to today

1. What Is a Pattern?

A pattern is a piece of literature that describes a design problem and a general solution for the problem in a particular context. Alexander tells us:

> *Each pattern is a three-part rule, which expresses a relation between a certain context, a problem, and a solution.*

I suggest a pronunciation of "al-ex-an-der-i-en" to distinguish it from the Alexandrian poetic form.

> The pattern is, in short, at the same time a thing, which happens in the world, and the rule which tells us how to create that thing, and when we must create it. It is both a process and a thing; both a description of a thing which is alive, and a description of the process which will generate that thing. (Alexander, 1979: p. 247)

I like to relate this definition to dress patterns. I could tell you how to make a dress by specifying the route of a scissors through a piece of cloth in terms of angles and lengths of cut. Or, I could give you a pattern. Reading the specification, you would have no idea what was being built or if you had built the right thing when you were finished. The pattern foreshadows the product: it is the rule for making the thing, but it is also, in many respects, the thing itself. The document that describes a particular decoupling arrangement in object-oriented programs, we call "the Bridge pattern" (Gamma et al., 1995); each instance of that arrangement in the program is also called "a Bridge." Also:

> These patterns in our minds are, more or less, mental images of the patterns in the world: they are abstract representations of the very morphological rules which define the patterns in the world.
>
> However, in one respect they are very different. The patterns in the world merely exist. But the same patterns in our minds are dynamic. They have force. They are generative. They tell us what to do; they tell us how we shall, or may, generate them; and they tell us too, that under certain circumstances, we must create them.
>
> Each pattern is a rule which describes what you have to do to generate the entity which it defines. (Alexander, 1979: pp. 181—182)

Above, Alexander refers to patterns as a "three-part rule"; he also calls patterns "rules of thumb" in *Timeless Way*. Patterns are more than just the rule: they encompass the literature that describes the rule and helps it unfold for our use.

Ward Cunningham adds the following metaphor—patterns are more like recipes than plans:

> I like to make the distinction between a plan and a recipe. A plan can be reverse engineered from a building but a recipe can't (easily) be reverse engineered from a cake. Our genome is a recipe, not a plan. Recipes seem to serve better as a schema of complex adaptive systems. (Personal communication with Ward Cunningham, 20 February 1996.)

We observe patterns in everyday structures: in buildings, in vehicle traffic, in organizations of people, and in our software. Alexander extracted patterns from building and town structures of numerous cultures. He believed that by documenting these patterns, he could help people shape the buildings of their community to support life to its fullest. Patterns are structures that have evolved over ages of development to fulfill cultural needs. They are rules, driven by principles, that serve human and social needs. Patterns are also the documentation of these structures. Therefore, a pattern is both "a thing which happens in the world," and "the rule... to create that thing."

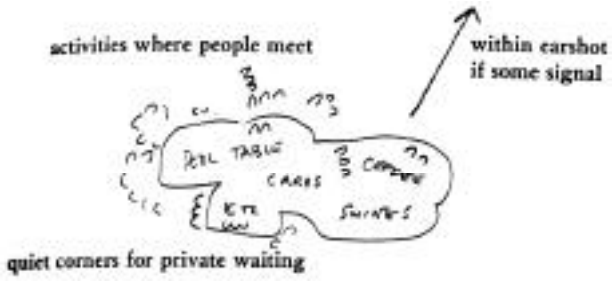
Patterns have evolved to a small number of related literary forms. In one sense, a pattern is just documentation. On the other hand, they are "just documentation" in the same sense that poetry is "just literature."

Besides, we believe that human communication is the bottleneck in software development. If the pattern literary form can help programmers communicate with their clients, their customers, and with each other, they help fill a crucial need of contemporary software development.

Patterns are not a complete design method; they capture important practices of existing methods and practices uncodified by conventional methods. They are not a CASE tool. They focus more on the human activities of

design than on transformations that can blindly be automated. They are not artificial intelligence; they celebrate and encourage the human intelligence that separates people from computers.

Example: Here is a synopsis of a typical Alexanderian pattern, one of the most often cited patterns from his book *A Pattern Language*:



A Place to Wait

The process of waiting has inherent conflicts in it.

One on hand, whatever people are waiting for— the doctor, an airplane, a business appointment— has built-in uncertainties, which make it inevitable that they must spend a long time hanging around, waiting, doing nothing.

On the other hand, they cannot usually afford to enjoy this time. Because it is unpredictable, they must hang at the very door. Since they never know exactly when their turn will come, they cannot even take a stroll or sit outside...

. . .

Therefore:

In places where people end up waiting, create a situation which makes the waiting positive. Fuse the waiting with some other activity—newspaper, coffee, pool tables, horseshoes; something which draws people in who are not simply waiting. And also the opposite: make a place which can draw a person waiting into a reverie; quiet; a positive silence. (Alexander et al., 1977: pp. 707—711)

We also see patterns in the software we write. Patterns are recurring design problem/solution pairs. The most important patterns capture important structures, practices, and techniques that are key competencies in a given field, but which are not yet widely known.

1.1. The Relationship of Patterns to...

Patterns are like many other design formalisms but are distinct from most common formalisms in subtle ways. Panu Viljamaa explains:

> Patterns are related to but different from: paradigms, idioms, principles, heuristics, architectures, frameworks, role-models.
>
> You could say that a paradigm is a very abstract pattern, or style of work that can be followed consistently throughout the system. Idiom is a language-specific typical way of using and combining elementary building blocks. Principle is an invariant that can hold globally, or “always”; it could be a synonym for “design rule.” Heuristics aid decision making, without claiming absolute goodness for the actions suggested. Heuristics could be used to choose among multiple alternative patterns. Architecture refers to the total structure of an application, possibly described by the multiple patterns involved. Patterns have been called “micro-architectures.” Frameworks refer to collections of concrete classes working together to accomplish a given parameterizable task. Role-models describe a single coordinated collaboration among multiple participants (the framework classes can serve in multiple roles simultaneously). Role-models may be the closest thing to the formalization of patterns. (Viljamaa, 1995)

1.2. Why Do We need to Capture Patterns?

Most would agree that the pattern discipline tries to capture important empirical design information. (The pattern community is certainly not the first to do this; for a discussion of past related efforts, see (Gamma et al., 1995: p. 357) Patterns have a different emphasis than most reuse programs or design catalogues: they tend to capture broader abstractions. In the spirit of Alexander's patterns, they make up for lapses in the memory of the contemporary software design culture, and they capture structure not immediately apparent from the code or from most system design documents.

Patterns capture obscure but important practice. Patterns capture established practices that remain obscure in the broad practice of a given domain. The intuitiveness of patterns is paradoxical. Many patterns have their roots in the work of early adaptors of a new technology or the first architects of a system. Many of these patterns attack problems in subtle ways, which makes it difficult to cast them in the framework of the predominate constructs of the technology or system. For example, in C++, reference counting would be a pattern, while the specific language features used to implement it are just that, language features, independent of reference counting per se. Bjarne Stroustrup foresaw the need for reference counting in C++. Many C++ language features—such as constructors and destructors (which serve other needs as well) and overloaded assignment—anticipate this need. But when people first learn C++, they learn it in terms of its language parts (classes, procedures, and objects) or in terms of object-oriented design principles that lead to a good class partitioning.

Patterns work at many levels of detail. We tend to think of reference counting as a detail, but programmers who don't know how to implement it properly will never be viable C++ programmers. Though reference counting is a low-level construct, we can use patterns to capture its key design principles in an abstract form. A pattern is abstract because it approaches the problem at a suitably general level, although the solution may entail details. A good solution has enough detail that the designer knows what to do, but it is general enough to address a broad context.

Patterns capture hidden structure. Patterns cur across the predominant partitionings of the subject area. Many Alexanderian patterns talk about the relationship between streets and houses (Entrance Transition), between houses and houses (Row Houses), between rooms and rooms (Varied Ceiling Heights), or between a room and its interfaces (Light on Two Sides of Every Room). They rarely focus on the properties of a single architectural artifact alone. Good software patterns are the same way: they address system problems and relationships that are obscured by a perspective from inside any of the parts. Early object-oriented design techniques magnified this myopic view of systems, and contemporary methods aren't much better. Patterns complement object-oriented design methods to capture the important constructs that cur across objects.

These "deep" components of architecture and design are larger than any architectural building blocks such as procedures or objects. Such constructs are often obscure to the day-to-day practitioner because the building materials—objects, modules, and procedures—don't highlight them. Some of these patterns are intricate and perhaps detailed—like reference counting or the structure of stairs in castle towers—yet such patterns permeate the structures they help create. The architect put such structures in place to fulfill a need, but such structures and the rationale behind them are soon lost to history. Patterns capture these structures and decisions.

Again, we can draw on Alexander:

> *Design is often thought of as a process of synthesis, a process of putting together things, a process of combination.*

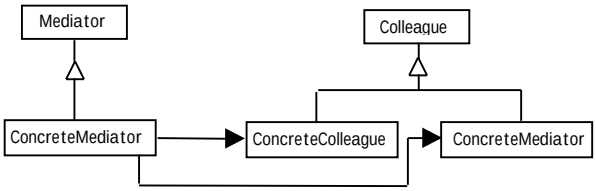
> According to this view, a whole is created by putting together parts. The parts come first: and the form of the whole comes second.
>
> *But it is impossible to form anything which has the character of nature by adding preformed parts*.
>
> When parts are modular and made before the whole, by definition then, they are identical, and it is impossible for every part to be unique, according to its position in the whole. (Alexander, 1979: p. 368)

Later in the same work:

> In order for the building to be alive, its construction details must be unique and fitted to their individual circumstances as carefully as the larger parts… The details of a building cannot be made alive when they are made from modular parts. (Alexander, 1979: pp. 459—460)

The Mediator pattern talks about the relationship between two objects, which often cuts across the intuitive object system partitioning:

> **Name**
>
> Mediator
>
> **Intent**
>
> Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.
>
> **Motivation**
>
> Object-oriented design encourages the distribution of behavior among objects. Such distribution can result in an object structure with many connections between objects; in the worst case, every object ends up knowing about every other.
>
> Though partitioning a system into many objects generally enhances reusability, proliferating interconnections tend to reduce it again. Lots of interconnections make it less likely that an object can work without the support of others—the system acts as though it were monolithic. Moreover, it can be difficult to change the system's behavior in any significant way, since behavior is distributed among many objects. As a result, you may be forced to define subclasses to customize the system's behavior.
>
> You can avoid these problems by encapsulating collective behavior in a separate **mediator** object. A mediator is responsible for controlling and coordinating the interactions of a group of objects. The mediator serves as an intermediary that keeps objects in the group from referring to each other explicitly. The objects only know the mediator, thereby reducing the number of interconnections.
>
> **Consequences**
>
> 1. *It limits subclassing...*
> 2. *It decouples colleagues...*
> 3. *It simplifies object protocols...*
> 4. *It abstracts how objects cooperate...*
> 5. *It centralizes control…*



(From Gamma et al., 1995: pp. 273—277)

1.3. The Pattern Form

Patterns are a literary form, much as sonnets or novels or short stories are literary forms. The form serves a purpose: to introduce the reader to a problem, to describe the context where the problem might arise, to analyze the problem, and to present and elucidate a solution.

Many documents classify as “patterns” within these guidelines, and the question of form is usually wrapped up with the definition of patterns. The prevailing definition: a solution to a problem in a context evokes elements of the form. There is large variety of established pattern forms. ‘The following section is a survey of forms in common use today. Section 1.3.1 describes sections of the form in more detail.

1.3.1. Sections of the Form

We can write a good pattern by ensuring the goals of each section are met. Writing a great pattern is more holistic than that. The pattern must work as a seamless piece of literature, but the sections and their attributes are worth investigation by the pattern student and pattern critic alike.

This section draws on ideas from a wide variety of forms in common use. It combines background and tips for each of the pattern sections. These sections also help put the pattern components in perspective. The solution is obviously the “heart of the pattern,” as Alexander said, but the sketch and the forces are unusually important, too. The common (or perhaps minimal) sections are:

- Name
- Intent
- Problem
- Context
- Forces
- Solution
- Sketch
- Resulting Context

Name

Names are important in many cultures, and have always loomed important in programming. A good object-oriented designer understands that it is important to choose good class names. Pattern names are important for at least two reasons. First, they are one of the first things a designer encounters when seeking a solution. If the name encodes the pattern’s meaning well, the designer can more easily find a suitable pattern in an unfamiliar pattern language. For example, “Ambassador” or “Remote Proxy” evoke powerful images of how two objects interact, and implementation descriptions are almost superfluous.

Second, pattern names quickly become part of the design team vocabulary. Short, well-chosen names aid communication between developers. For example, Riding Over Transients (page 26) is a general pattern that deals with transient events: if the designer is looking for a solution to transient errors, the name gets the designer’s attention and encourages further reading. Half-Object Plus Protocol, Window Per Task, and Exceptional Value are other descriptive pattern names.

Many patterns take clever names that recall obscure analogies or historically relevant folklore. We find pattern names and pattern language names like “Leaky Bucket Counter,” “Fool Me Once,” and “Caterpillar’s Fate” that convey deep meaning once explained. Perhaps because these names have strong cultural ties, they are great communication enablers, but such names are a liability to novice designers who are unfamiliar with the culture. Novices seek names that describe the problem or its solution.

Aliases can help address this problem. Leaky Bucket Counter might have an alias of Riding over Event Transients: the first name evokes a powerful analogy for those who know the pattern, and the second name serves the novice solution seeker.

Should a pattern title be a noun, a verb, a noun phrase, or a verb phrase? Should it characterize the problem or the solution? We find patterns that follow each of these and then

some. Alexander's pattern names are predominately nouns or noun phrases, with minor exceptions, but we find many variations that work well. Here are a few noun phrases:

- A Place to Wait (Alexander, 1977: pp. 707—711). Describes the context for the solution.
- Capacity Bottlenecks (Meszaros, 1996). Describes the problem.
- Bridge (Gamma et al., 1995: pp. 151—153). Describes the solution.
- Chicken and Egg (see page 22). Describes the problem.
- Gatekeeper (Coplien, 1995a). Describes a key component of the solution.

Here are so me verb phrases:

- Fool Me Once and Minimize Human Intervention (Adams, et al., 1996). Clever names for the solutions.
- Developing in Pairs (Coplien, 1995a). Describes a solution.
- Reception Welcomes You (Alexander, 1977: pp. 705—706). A rare Alexanderian title with a prominent verb; it describes a solution.
- Identify the Nouns (DeBruler, 1995). Describes the problem.

Intent, Question

The intent is a phrase or sentence that summarizes what the pattern does, describing the design issue or problem it addresses. As a designer scans patterns for a solution to a specific problem, the intents provide a road map into promising patterns and around irrelevant ones.

The intent often is not part of the pattern body per se, but it may be used to annotate the pattern index. That makes it easier for designers to quickly find a pattern that meets their need.

The Intent section was first used in the "Gang of Four (GOF)" pattern form (page 12) but has a direct analogy in the question that appears an the beginning of many pattern forms (e.g., the questions at the beginning of Kent Beck's patterns written in the Portland Form, page 13).

Problem

The problem section describes the problem to be solved. A concise problem statement helps the problem solver decide whether to read further, and it often serves as the primary index for pattern selection.

Some pattern forms reduce the problem to a single question or a summary formulation of the problem; many of the patterns of the Coplien form (page 14) have this feature. In other forms, the problem statement is a small essay that motivates or illustrates a need. For example, Alexanderian form (page 12) combines the forces into the problem section.

Context

We make the context explicit in a pattern. Context includes a history of patterns that have been applied before the current pattern was considered. It also specifies size, scope, market, programming language, or anything else that, if changed, would invalidate the pattern.

Pattern context is crucial to the success of a pattern language, a collection of patterns that work together to solve system-level problems (see Section 2). One can think of a pattern as balancing forces for a problem in one context, leaving a new context. Contexts weave patterns together into a pattern language.

It is difficult to write a good context. The context section matures with experience: as designers find special situations that invalidate the pattern, the context grows to become more restrictive. Sometimes, we find opportunities to grow the pattern into new contexts, and capture that experience in the context section.

As patterns evolve, the pattern writer should take care to maintain a lucid context section. Domain vocabularies evolve with increased understanding, and one might be able to make the Context more precise only at the expense of accessibility.

Forces

Patterns aren't rules we follow blindly; we should understand them and tailor them to our needs. If we understand the forces in a pattern, then we understand the problem (because we understand the trade-offs) and the solution (because we know how it balances the forces), and we can intuit much of the rationale. For this reason, forces are the focus of a pattern. Alexander's early writings emphasize that forces are crucial to understanding systems:

> No one will become a better designer by... following any method blindly…if you try to understand the idea that you can create abstract patterns by studying the implication of limited systems of forces, and can create new forms by free combination of these patterns—and realize that this will only work if the patterns which you define deal with systems of forces whose internal interaction is very dense, and whose interaction with the other forces in the world is very weak—then, in the process of trying to create such diagrams or patterns for yourself, you will reach the central idea of which this book is all about. (Alexander, 1974, Preface to the Paperback Edition)

From a practical perspective, forces help the designer understand how to apply a pattern effectively. A single pattern can be applied a million times without ever doing quite the same thing twice. The key to these distinctions lies in the forces.

The term *force* appeals to the architectural heritage of patterns. A building architect designs arches and walls to balance the forces of gravity with the forces from adjoining structures, so the structure is balanced and centered. Balanced forces support a firm, structurally sound system. In software, we use the term *force* figuratively because there are rarely physical forces we must balance. Even Alexander used the force in a figurative sense, particularly as he was concerned about balancing the forces of human aesthetics and comfort with the physical structure of a town or building. Great architects can balance all of these (i.e., they don't optimize aesthetics at the expense of implementation feasibility, utility, or cost—a practicality that seems to be lost in many modern architecture schools). As patterns mature, they move past purely technical and mechanical forces and take human forces into account.

The forces should amplify and illustrate the problem statement because it is through the forces that one fully appreciates the problem. In Alexanderian form (page 12), the forces are part of the problem statement proper—not a separate section. Other forms separate out the forces to help point the reader to the exposition of trade-offs.

Forces determine why a problem is difficult. If a designer understands the forces of the pattern, the soundness of the solution becomes obvious. Describing what we understand not to work points the way to what works. Alexander says:

> We should find it almost impossible to characterize a house which fits its context. Yet it is the easiest thing in the world to name the specific kinds of misfit which prevent good fit. A kitchen which is hard to clean, no place to park my car, the child playing where it can be run down by someone else's car, rainwater coming in, over-

> crowding and lack of privacy, the eye-level grill which spits hot fat right into my eye, the gold plastic doorknob which deceives my expectations, and the front door I cannot find, are all misfits between the house and the lives and habits it is meant to fit. These misfits are the forces which must shape it, and there is no mistaking them. Because they are expressed in negative form they are specific, and tangible enough to talk about. (Alexander, 1974: pp. 22—23)

Consider this pattern from Gerard Meszaros, particularly the deliciously disturbing forces:

> **Leaky Bucket of Credits**
>
> Problem:
>
> How can one processor know whether another processor is capable of handling more work?
>
> Forces:
>
> For a peripheral to be able to reject new work when the system is overloaded, it must be able to recognize when the system is overloaded. But having the bottleneck processor take up valuable realtime cycles to inform the (potentially large number of) peripherals would further reduce its capacity. And what happens if it gets so bogged down that it can't send out the "Stop sending me work!" messages?
>
> Solution:
>
> The bottleneck processor tells the peripherals when it is capable of accepting more work. It does so by sending "credits" to the peripherals. Each peripheral tracks a leaky bucket of credits received from the bottleneck processor [(BP)]. As requests are sent to the BP, or simply as time passes, the bucket leaks until it is empty. When the system is not at capacity, the bucket is continuously refilled by new credits sent from the BP; however, if the system is at capacity, the BP will not send credits and the peripherals will hold back new work... (Meszaros, 1996)

The forces, taken alone, inform the reader that this is a hard problem—perhaps an intractable problem! The forces tug the design in two directions, helping our thought process to explore all the options and to consider the dark corners of the design. The solution resolves the forces, but the solution means more for the forces having "set it up."

Solution

A good solution has enough detail so the designer knows what to do, but it is general enough to address a broad context.

The solution should solve the problem stated in the problem section. Some patterns provide only partial solutions but open a path to other patterns that balance unresolved forces.

If a pattern is literature, it is like a play in that the solution section should provide catharsis. The context introduces the "characters" and the "setting"; the Forces provide a "plot," and the Solution provides a resolution of the tension built up in the conflict of the Forces. This form helps emphasize the importance of the solution.

Sketch

Alexander maintained that the sketch is the essence of a pattern. In *Notes on the Synthesis of Form*, one of his earliest works, he notes:

> These diagrams, which, in my more recent work, I have been calling patterns, are the key to the process of creating form.., most of the power of what I had written lay in the power of these diagrams...
>
> Poincaré once said: 'Sociologists discuss sociological methods; physicists discuss physics.' I love this statement. Study of method by itself is always barren, and people who have treated [Notes on Synthesis] as if it were a book about 'design method' have almost always missed the point of the diagrams, and their great importance, because they have been obsessed with the

> details of the method I propose for getting an the diagrams. (Alexander, 1974: p. ii)

And in *The Timeless Way of Building*, he says, simply:

> If you can't draw a diagram of it, it isn't a pattern. (Alexander, 1979: p. 267)

What do we sketch in a software pattern? Anything we think helps the designer understand the relationship between the parts. The sketch usually conveys *structure*. The *Design Pattern* book (Gamma et al., 1995) uses OMT diagrams (a widely accepted software design notation) to present example solution structures for each of its patterns. Interaction diagrams that illustrate event ordering or other dynamics are equally useful.

Classical architects did not draw on abstract notions of art for its own sake, but limited themselves to familiar structures that served the culture well. In that sense, architecture is unlike painting. And the builders of comfortable folk architecture homes usually do not follow a prescribed grand plan, but assemble mud, thatch, stone, or brick to build the foundation first, the walls and doorway second, the windows next, then the roof, and finally the interior walls. Preordaining all these structures makes it impossible for the builder to manage the interplay between a growing house and its environment: light, wind, and the shadows of neighboring structures. Even if one can foresee such problems, the initial two-dimensional drawings can never capture the magic (or funkiness, as the case may be) of the completed structure.

Software specifications and architecture documents are analogous to construction blueprints. Consider Alberti's Law:

> Here is another liability: beautiful drawings can become ends in themselves. Often, if the drawing deceives, it is not only the viewer who is enchanted but also the maker, who is the victim of his own artifice. Alberti understood this danger and pointed out that architects should not try to imitate painters and produce lifelike drawings. The purpose of architectural drawings, according to him, was merely to illustrate the relationship of the various parts... Alberti understood, as many architects of today do nor, that the rules of drawing and the rules of building are not one and the same, and mastery of the former does not ensure success in the latter. (Rybczynski, 1989: p. 121)

Alexander says much the same thing:

> It is essential, therefore, that the builder build only from rough drawings: and that he carry out the detailed patterns from the drawings according to the processes given by the pattern language in his mind. (Alexander, 1979: p. 463)

This is why the sketch is called a "sketch" and not a "graphical specification." Most readers interpret refined diagrams too literally. There is much to be said for hand-drawn diagrams that abhor right angles and straight lines (see, for example, the sketches of Detached Counted Handle/Body idiom on page 23). Such a rough solution encourages the designer to craft or engineer the solution to the situation at hand.

Resulting Context

The Resulting Context is the wrap-up of the pattern. It tells us:

- which forces were resolved
- which new problems may arise because of this pattern
- what related patterns may come next

Each pattern is designed to transform a system in one context to a new context. The Resulting Context of one pattern is input to the patterns that follow. Contexts tie related patterns together into a pattern language (Section 2).

1.3.2. Common Forms

A pattern is a literary form. We have seen several forms emerge and evolve over the past four years. This section summarizes the most popular pattern forms, drawing directly on their originators' insights.

Alexanderian Form

Alexanderian form, from Christopher Alexander's work, is the "original" pattern form. The sections of an Alexanderian pattern are not strongly delimited. The major syntactic structure is a *Therefore* immediately preceding the solution. Other elements of the form are usually present: a clear statement of the problem, a discussion of forces, the solution, and a rationale.

Each Alexanderian pattern usually follows an introductory paragraph that enumerates the patterns that must already have been applied to make the ensuing pattern meaningful. The pattern itself starts with a name and a confidence designation of zero, one, or two stars. Patterns with two stars are the ones in which the authors have the most confidence because they have empirical foundations. Patterns with fewer stars may have strong social significance but are more speculative.

Here is Alexander's own description of his form:

> For convenience and clarity, each pattern has the same format. First, there is a picture, which shows an archetypal example of that pattern. Second, after the picture, each pattern has an introductory paragraph, which sets the context for the pattern, by explaining how it helps to complete certain larger patterns. Then there are three diamonds to mark the beginning of the problem. After the diamonds there is a headline, in bold type. This headline gives the essence of the problem in one or two sentences. After the headline comes the body of the problem. This is the longest section. In describes the empirical background of the pattern, the evidence for its validity, the range of different ways the pattern can be manifested in a building, and so on. Then, again in bold type, like the headline, is the solution—the heart of the pattern—which describes the field of physical and social relationships which are required to solve the stated problem, in the stated context. This solution is always stated in the form of an instruction—so that you know exactly what you need to do, to build the pattern. Then, after the solution, there is a diagram, which shows the solution in the form of a diagram, with labels to indicate its main components.
>
> After the diagram, another three diamonds, to show that the main body of the pattern is finished. And finally, after the diamonds there is a paragraph which ties the pattern to all those smaller patterns in the language, which are needed to complete this pattern, to embellish it, to fill it out. (Alexander et al., 1977: pp. x—xi)

Why does Alexander adopt this form? He goes on:

> There are two essential purposes behind this format. First, to present each pattern connected to other patterns, so that you grasp the collection of all 253 patterns as a whole, as a language, within which you can create an infinite variety of combinations. Second, to present the problem and solution of each pattern in such a way that you can judge it for yourself, and modify it, without losing the essence that is central to it. (Alexander et al., 1977: p. xi)

The Detached Counted Handle/Body idiom (page 23) and Simply Understood Code (page 40) are examples of software patterns written in Alexanderian form.

The GOF Form

The GOF ("Gang of Four") Form was established in *Design Patterns* (Gamma et al., 1995). It has the following sections:

> Pattern Name and Classification: The pattern's name conveys the essence of the pattern succinctly. A good

name is vital, because it will become part of your design vocabulary...

Intent: A shorn statement that answers the following questions: What does the design pattern do? What is its rationale and intent? What particular design issue or problem does it address?

Also Known As: Other well-known names for the pattern, if any.

Motivation: A scenario that illustrates a design problem and how the class and object structures in the pattern solve the problem. The scenario will help you understand the more abstract description of the pattern that follows.

Applicability: What are the situations in which the design pattern can be applied? What are examples of poor designs that the pattern can address? How can you recognize these situations?

Structure: A graphical representation of the classes in the pattern using a notation based on the Object Modeling Technique (OMT) [Rumbaugh et al., 19911. We also use interaction diagrams (Jacobson et al., 1992; Booch, 1994) to illustrate sequences of requests and collaborations between objects...

Participants: The classes and /or objects participating in the design pattern and their responsibilities.

Collaborations: How the participants collaborate to carry out their responsibilities.

Consequences: How does the pattern support its objectives? What are the trade-offs and results of using the pattern? What aspect of system structure does it let you vary independently?

Implementation: What pitfalls, hints, or techniques should you be aware of when implementing the pattern? Are there language-specific issues?

Sample Code: Code fragments that illustrate how you might implement the pattern in C++ or Smalltalk.

Known Uses: Examples of the pattern found in real systems. We include at least two examples from different domains.

Related Patterns: What design patterns are closely related to this one? What are the important differences? With which other patterns should this one be used? (Gamma et al., 1995: pp. 6—7)

The GOF pattern form is tuned for object-oriented software designs. The Mediator pattern (page 6) and the Bridge pattern (page 15) are examples of GOF patterns.

The Portland Form

Ward Cunningham maintains an on-line repository of patterns called the Portland Pattern Repository. Many of the patterns found in that forum follow the Portland Form, of which Cunningham writes:

> The repository prefers patterns written in the Portland Form, a form first adopted by three authors submitting papers to the Pattern Languages of Programs conference, PLoP ‘94. (All three were from Portland, Oregon, hence the name.) The form has been described as narrative, as opposed to the more outline like form of keyword templates first used by Peter Coad and made popular by Erich Gamma et al. The form is actually a fairly direct emulation of Alexander’s form with some simplification in typesetting. We hope that the hypertext aspects of the repository will more than make up for the omissions and simplifications of the Portland Form.
>
> **Language Document**
>
> Each document in the Portland Form contains a system of patterns that work together. Alexander calls such systems “languages” since he believes the human mind assembles the words of a natural language, namely, without much conscious thought. The Portland Form collects and connects patterns so that they will be studied and understood as a whole. Although we believe all patterns will be ultimately linked, we cur-

rently give authors the responsibility of defining a suitable whole consistent with their own knowledge and their readers' ability to absorb. This unit we call both a language and a document...

Pattern Paragraphs

Each pattern in the Portland Form makes a statement that goes something like: "such and so forces create this or that problem, therefore, build a thing-a-ma-jig to deal with them." The pattern takes its name from the thing-a-ma-jig, the solution. Each pattern in the Portland Form also places itself and the forces that create it within the context of other forces, both stronger and weaker, and the solutions they require. A wise designer resolves the stronger forces first, then goes on to address weaker ones. Patterns capture this ordering by citing stronger and weaker patterns in opening and closing paragraphs. The total paragraph structure ends up looking like:

- Having done so and so you now face this problem...
- Here is why the problem exists and what forces must be resolved...

Therefore:

- Make something along the following lines. I'll give you the help I can...
- Now you are ready to move on to one of the following problems...

Summary Screen

Long pattern languages find groups of patterns working around similar ideas. Portland Form introduces such groups with a summary section. This section explains the general problem under consideration and names the patterns that address it. (Source: The Portland Pattern Repository, http:// c2.com/ppr/)

The CHECKS patterns (page 41) are examples of the Portland form.

The Coplien Form

The Coplien form also reflects the basic elements found in the Alexanderian form. It delineates pattern sections with section headings and includes:

- **The pattern name**: The Coplien form commonly uses nouns for pattern names, but short verb phrases can also be used. This follows from the Alexanderian form.
- **The problem**: The problem is often stated as a question or design challenge. This is analogous to the Alexanderian section that follows the first three diamonds.
- **The context**: A description of the context in which the problem might arise, and to which the solution applies. This is like Alexander's introductory paragraph that sets context.
- **The forces**: The forces describe pattern design trade-offs; what pulls the problem in different directions, toward different solutions? This is like Alexander's in-depth description of the problem, the longest part of the pattern.
- **The solution**: The solution explains how to solve the problem, just as in the emboldened section of an Alexanderian pattern. A sketch may accompany the solution—analogous to the second sketch of Alexander's patterns.
- **A rationale**: Why does this pattern work? What is the history behind the pattern? We extract this so it doesn't "clutter" the solution. As a section, it draws attention to the importance of principles behind a pattern; it is a source of learning, rather than action.
- **Resulting context:** This tells which forces the pattern resolves and which forces remain unresolved by the pattern, and it points to more patterns that might be the next ones to consider. This is like the Alexanderian section following the second set of three diamonds.

The Counted Body Idiom (Section 1.3.3) is in Coplien form.

1.3.3. Relationships between Parts of a Pattern

A pattern's solution section balances its problem section. However, several patterns may address the same problem, and a single solution may address multiple problems.

As an example, consider the following idiom (a low-level pattern) called Counted Body:

> **Name:** Counted Body Idiom
>
> **Problem:** Simulating Smalltalk assignment semantics in C++
>
> **Context:** A design has been transformed into body/handle C++ class pairs.
>
> **Forces:**
>
> Assignment in C++ is defined recursively as member-by-member assignment with copying as the termination of the recursion; it would be more efficient and more in the spirit of Smalltalk if copying were rebinding.
>
> Deep copying of bodies is expensive.
>
> Copying can be avoided by using pointers and references, but these leave the problem of who is responsible for cleaning up the object, and they leave a user-visible distinction between built-in types and user-defined types.
>
> Sharing bodies on assignment is semantically incorrect if the shared body is modified through one of the handles.
>
> **Solution:** A reference count is added to the body class to facilitate memory management.
>
> Memory management is added to the handle class, particularly to its implementation of initialization, assignment, copying, and destruction.
>
> It is incumbent on any operation that modifies the state of the body to break the sharing of the body by making its own copy. It decrements the reference count of the original body.
>
> **Forces Resolved:**
>
> Gratuitous copying is avoided, leading to a more efficient implementation.
>
> Sharing is broken when the body state is modified through any handle.
>
> Sharing is preserved in the more common case of parameter passing, etc.
>
> Special pointer and reference types are avoided. Smalltalk semantics are approximated; garbage collection is driven off of this model.
>
> **Design Rationale**: Reference counting is efficient and spreads the overhead across the execution of real-time programs.

Compare this pattern with the pattern Detached Counted Handle/Body Idiom (see page 23). Both solve the same problem: memory management. Both solve the problem by separating the implementation from the interface. Why are they different patterns? Each has a different context, and we find these contexts elucidated in the forces. In "Detached Counted Handle/Body," we find that the class to be managed is a library class, to which we cannot add a reference count member. In its resulting context, we find concerns about memory overhead and fragmentation—less of a problem in the ordinary Handle/Body idiom (which is like the Counted Handle/Body idiom, but without reference counting).

Now compare Gamma et al.'s Bridge pattern (Gamma et al., 1995):

> **Intent**
>
> Decouple an abstraction from its implementation so that the two can vary independently.

Also Known As

Handle/Body

Motivation

When an abstraction can have one of several possible implementations, the usual way to accommodate them is to use inheritance. An abstract class defines the interface to the abstraction, and concrete subclasses implement it in different ways. But this approach isn't always flexible enough. Inheritance binds an implementation to the abstraction permanently, which makes it difficult to modify, extend, and reuse abstractions and implementations independently.

The Bridge pattern addresses these problems by punning the ... abstraction and its implementation in separate class hierarchies.

Applicability

Use the Bridge pattern when

- You want to avoid a permanent binding between an abstraction and its implementation...
- Both the abstractions and their implementations should be extensible by subclassing...
- Changes in the implementation of an abstraction should have no impact on clients; that is, their code should not have to be recompiled.
- (C++) you want to hide the implementation of an abstraction completely from clients...
- You have a proliferation of classes... Such a class hierarchy indicates the need for splitting an object into two parts...
- You want to share an implementation among multiple objects (perhaps using reference counting), and this fact should be hidden from the client. A simple example is Coplien's String class [Coplien, 1992], in which multiple objects can share the same string representation (StringRep). (Gamma et al., 1995: pp. 151—153)

These patterns are clearly related. However, the Bridge pattern describes the problem from the perspective of the solution: the need to separate implementation from interface; reference counting is at the end of the Applicability list.

The problem solved by the Counted Body idiom is to simulate Smalltalk assignment semantics in C++. The Detached Counted Handle/Body Idiom solves a similar problem: it gives classes the same reasonable behavior we come to expect from built-in types. In many contexts, the solution for that problem is to separate interface from implementation, which is the key problem solved by Bridge.

Many of these subtleties are germane to the imprecision of natural language and the complexity of design. Hopefully, future research will provide better foundations for a "linguistics of patterns" that helps regularize the relationship between the problem and solution space.

1.4. Patterns and Paradigms

One of the most important challenges of system design is dealing with complexity. We attack complexity with abstraction. Much of design concerns itself with finding the "right" abstractions in a system, partitioning the system into mind-size, manageable chunks.

To divide a system into parts, we usually use a consistent set of guidelines, principles, rules, and tools. A paradigm is a world view that embraces a consistent set of rules and tools that we use to partition a system into manageable abstractions. It gives us broad organizing principles that help us structure systems from a very broad understanding of their functionality and markets in the possible absence of domain experience or expertise.

There are holes in the broad space of design that most paradigms and their design methods leave unfilled. Great designers know how to fill those holes by drawing on their intuition or on past experience.

Those are the kind of design insights we want to capture in patterns: design structures that can't easily be regularized in a method.

As technology progresses, today's patterns become tomorrow's paradigms; tomorrow's paradigms become next week's programming language. The design tricks known to early compiler designers are now regularized in tools like **yacc** and **bison**. Some patterns seem to defy regularization even over time. Many of the patterns of classic telecommunication are still part of the design folklore. There is a cost associated with keeping such knowledge locked up as folklore: it makes it difficult to (re)-staff and maintain legacy systems. Patterns strive to bring such knowledge into the open.

2. What Are Pattern Languages?

A pattern language is a collection of patterns that build on each other to generate a system. A pattern in isolation solves an isolated design problem; a pattern language builds a system. It is through pattern languages that patterns achieve their fullest power.

The term pattern language comes from building architecture and has been popularized by Alexander. We can compare a pattern language to natural language. English can generate all meaningful English sentences, and a pattern language that deals with data form input errors (like the CHECKS pattern language; see page 41) can generate the error-handling architecture for all human interfaces that fin the context.

This sense of the word language is not commonly used in the computer field. A pattern language should not be confused with a programming language. A pattern language is a piece of literature that describes an architecture, a design, a framework, or other structure. It has structure, but not the same level of formal structure that one finds in programming languages. The term pattern language has been the source of some confusion because of this, leading some authors to instead use the term "pattern system (Buschmann & Meunier, 1995).

Pattern languages are closely related to Parnas' notion of a software family (Parnas, 1976). A software family comprises many members related to each other by their commonality, distinguishable from each other by their variability. Think of a pattern language as the collection of rules that build all members of a family and only members of a family. Cape Cod Houses come from a pattern language; Alexander presents pattern languages for farmhouses in the Bernese Oberland, for southern Italian stone houses, and for other genres (Alexander, 1979: p. 185).

A pattern language is not just a decision tree of patterns. This is partly because the patterns of a pattern language form a directed acyclic graph (DAG), not a hierarchy. The number of distinct paths through a pattern language is very large.

Pattern languages place individual patterns in context. Alexander says:

> *Each pattern then, depends both on the smaller patterns it contains, and on the larger patterns within which it is contained...*
>
> . . .
>
> *And it is the network of these connections between patterns which creates the language.*
>
> . . .
>
> *In this network, the links between the patterns are almost as much a part of the language as the patterns themselves.*
>
> . . .
>
> *It is, indeed, the structure of the network which makes sense of individual patterns, because it anchors them, and helps make them complete.*
>
> . . .

But even when I have the patterns connected to one another, in a network, so that they form a language, how do I know if the language is a good one?

. . .

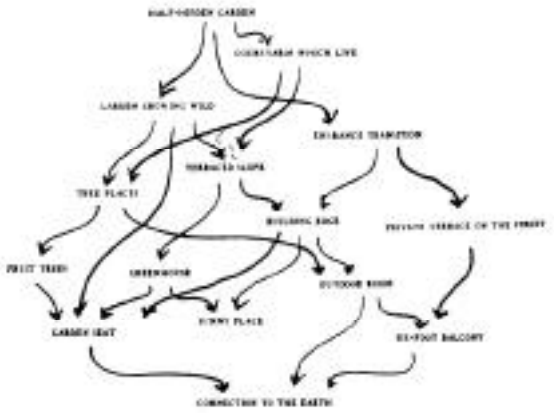
The language is a good one, capable of making something whole, when it is morphologically and functionally complete.

. . .

The language is morphologically complete when I can visualize the kind of buildings which it generates very concretely

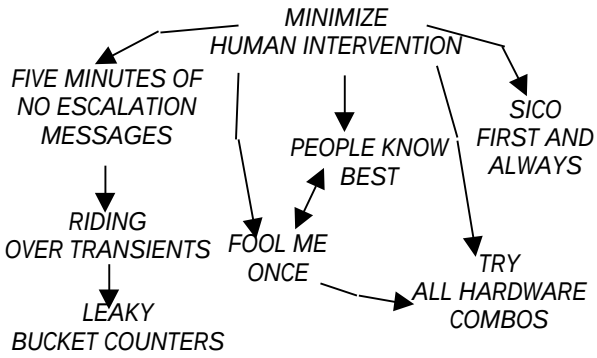
. . .

And the language is functionally complete, when the system of patterns it defines is fully capable of allowing all its inner forces to resolve themselves. (Alexander, 1979, pp. 312—317)



2.1. An Example Software Pattern Language

AT&T has assembled a large pattern language that captures expert practices from one of its key business domains. That large language can be broken down into many self-consistent languages of more modest scope. One such language was published by Adams, et al., (1996); it includes the patterns Leaky Bucket Counter and Riding Over Transients (page 26) as small patterns that complement the pattern Minimize Human Intervention. The structure of the pattern language looks like the following figure (adopted from Adams, et al., 1996):



We'll examine the two patterns in the lower left of the diagram in more detail later (see page 26). Let's look at Five Minutes of No Escalation Messages:

Name: Five Minutes of No Escalation Messages

Problem: Rolling in console messages: the human-machine interface is saturated with error reports that may be rolling off the screen or consuming resources just for the intense displaying activity.

Context: Any continuous-running, fault-tolerant system with escalation, where transient conditions may be present.

Forces: There is no sense in wasting time or in reducing the level of service trying to solve a problem that will go away by itself.

Many problems work themselves out given time.

You don't want the switch using all of its resources displaying messages.

You don't want to panic the user by making them think the switch is out of control (Minimize Human Intervention).

The only user action related to the escalation messages may be inappropriate to the goal of preserving system sanity.

There are other computer systems monitoring the actions taken. These systems can deal with a great volume of messages.

Solution: When taking the first action down the scenario that could lead to an excess number of messages, display a message. Periodically display an update message. If the abnormal condition ends, display a message that everything is back to normal. Do not display a message for every change in state (Riding Over Transients).

Continuously communicate status and actions taken to the downstream monitoring computer system throughout this period.

For example, when the 4ESS™ Switch enters the first level of system overload, post a user message. Post no more messages for 5 minutes, even if there is additional escalation. At the end of 5 minutes, display a status message indicating the current status. When the condition clears, display an appropriate message.

Resulting Context: The system operator won't panic from seeing too many messages. Machine-to-machine messages and measurements will keep a record for later evaluation as well as keeping the system actions visible to people who can deal with them. In the 4ESS overload example, measurement counters continue to track overload dynamics; some downstream support systems track these counters.

Other messages, not related to the escalating situation that is producing too many messages, will be displayed as though the system were normal. Thus the normal functioning of the system is not adversely affected by the volume of escalation messages.

Note the conflict with People Know Best.

(Adams, et al., 1996)

This pattern calls on Riding Over Transients as one of the primary implementation mechanisms; Riding Over Transients in turn employs Leaky Bucket Counter. We might use a Leaky Bucket Counter to implement Five Minutes of No Escalation Messages for variations on "five minutes."

What makes these patterns a language is that they call on each other and work together to solve a broad problem, in this case, Minimize Human Intervention with the system. Many of the patterns in this language appear in other languages as well. Leaky Bucket Counters is a versatile pattern that can work at many levels.

A single pattern solves a single problem in a given, general way. Different contexts may suggest substantially different solutions to what is otherwise the same problem. Each of these contexts deserves a new pattern. A pattern language may present multiple patterns for the same problem, although each solution suits a different context. People Know Best attacks the reliability problem much differently than the other patterns in this pattern language:

Name: People Know Best

Problem: How do you balance automation with human authority and responsibility?

Context: High-reliability continuous-running systems, where the system itself tries to recover from all error conditions.

Forces: People have a good subjective sense of the passage of time, and how it relates to the probability of a serious failure, or how it will be perceived by the customer.

The system is set up to recover from failure cases (Minimize Human Intervention).

People feel a need to intervene.

Most system errors can be traced to human error.

Solution: Assume that people know best, particularly the maintenance folks. Design the system to allow knowledgeable users to override the automatic controls.

Example: [related to the pattern Try All Hardware Combos]...

Resulting Context: People feel empowered; however, they are also responsible for their actions.

This is an absolute rule: people feel a need to intervene. There is no perfect solution for this problem, and the pattern cannot resolve all the forces well. Fool Me Once is a partial solution, in that it doesn't give the human a chance to intervene.

Notice the tension between this pattern and Minimize Human Intervention. (Adams et al., 1996)

Notice that the pattern makes the tension explicit and that it points to the alternative pattern (which applies to a different context). What does Minimize Human Intervention look like?

Name: Minimize Human Intervention

Problem: History has shown that people cause the majority of problems in continuously running systems (wrong actions, wrong systems, wrong buttons).

Context: High-reliability continuous-running digital systems, where downtime, human-induced or otherwise, must be minimized.

Forces: Humans are truly intelligent; machines aren't. Humans are better an detecting patterns of system behavior, especially among seemingly random occurrences separated by time (People Know Best).

Machines are good an orchestrating a well-thought-out, global strategy, and humans aren't.

Humans are fallible; computers are often less fallible.

Humans feel a need to intervene if they can't see that the system is making serious attempts at restoration. Human reaction and decision times are very slow (by orders of magnitude) compared to computer processors.

A quiet system is a dead system.

Human operators get bored with ongoing surveillance and may ignore or miss critical events.

Normal processing or failure events are happening so quickly that inclusion of the human operator is infeasible.

Solution: Len the machine try to do everything itself, deferring to the human only as an act of desperation and last resort.

Resulting Context: A system less susceptible to human error. This will make the system customers happier. In many administrations, the system operator's compensation is based on system availability, so this strategy actually improves the lot of the operator.

Rationale: Empirically, a disproportionate fraction of high-availability system failures are operator errors, not primary system errors. By minimizing human intervention, the overall system availability can be improved. Human intervention can be reduced by build-

> ing in strategies that counter human tendencies to act rashly; see patterns like Fool Me Once, Leaky Bucket Counters, and Five Minutes of No Escalation Messages.
>
> Note the tension between this pattern and People Know Best. (Adams et al., 1996)

This top-level pattern clearly points to the smaller patterns that refine and complete it for specific contexts. Such pattern relationships make this a language with much more structure than a collection of loosely related patterns.

Many of the patterns in this language point to People Know Best as an outlier, yet the pattern language makes it clear that the pattern can't be ignored. Each pattern tells us to do something specific, but a total design requires a balanced application of all of these patterns together. No two systems built using these patterns will be the same because the systems depend on the design tradeoffs of the business and the application that serves it.

This language as a whole generates a family of architectures. Most modern telecommunication systems use these or closely related patterns. Thee patterns define a software genre, not in terms of protocols, interfaces or requirements, but in terms of the basic structures and mechanisms that arise from business needs. The lives both of customers and of the people who maintain these systems are improved by these patterns. Companies are starting to recognize the pattern languages behind their systems and are using them for documentation and system construction (see Section 5 or the reference by Beck et al. [1996]).

2.2. Pattern Languages Compared to Pattern Catalogues

The popular patterns of Gamma et al. (1995) form a pattern catalogue, but they do not form a pattern language.

The patterns are not complete enough to generate all programs in a domain (object-oriented programs are the domain of the Gamma et al. patterns).

This doesn't mean that the patterns in a catalogue are without structure. Gamma et al. portray the structure of their pattern catalogue in the book (inside the back cover); others have described additional taxonomies for the Gamma patterns. (Zimmer, 1995)

Pattern catalogues, particularly the work of Gamma et al., are the most common source of patterns in contemporary use. The book by Gamma et al. was timely: lane enough that it captured proven practice of the object paradigm and early enough to be useful to the enormous body of emerging object-oriented practitioners. Such basic patterns are likely to remain a key part of software literature for years to come because their problems and solutions are timeless. Yet we can go even further: These patterns don't rise to system concerns. We will likely see more and more pattern languages in narrow application domains, such as client/server design, distributed financial transaction processing, fault-tolerant telecommunications, and many more. Although a manageable pattern language can't support all aspects of system design in general, it can do a good job within a well-constrained domain. Pattern languages will continue to grow in importance as the discipline matures.

3. Pattern Domains

Software patterns nook their cue from Alexander's patterns of urban design and building architecture. We have adapted his principles and values (but not the patterns themselves) to software production. The leap from building architecture to program architecture is intuitive to many: both concern structure. But patterns can and have been used in less structured software disciplines such as development process and training. Some pattern languages (e.g., Berczuk, 1996b) combine multiple domains.

Here is a process pattern from Coplien and Chisholm that captures a recurring practice of high-performance software teams:

> **Pattern Name: Mercenary Analyst**
>
> ***Problem***: Supporting a design notation and the related project documentation is too tedious a job for people directly contributing to product artifacts.
>
> ***Context***: You are assembling the roles for the organization. The organization exists in a context where external reviewers, customers, and internal developers expect to use project documentation to understand the system architecture and its internal workings. (User documentation is considered separately).
>
> ***Forces***:
>
> If developers do their own documentation, it hampers "real" work.
>
> Documentation is often write-only.
>
> Engineers often don't have good communication skills.
>
> Architects can become victims of the elegance of their own drawings (see rationale).
>
> ***Solution***: Hire a technical writer who is proficient in the necessary domains but who has no stake in the design itself. This person will capture the design using a suitable notation and will format and publish the design for reviews and for consumption by the organization itself
>
> The documentation itself should be maintained online where ever possible. It must be kept up-to-date (therefore, *Mercenary Analyst* is a full-time job), and it should relate to customer scenarios (as in the pattern Scenarios Define Problem).
>
> ***Resulting Context*:** The success of this pattern depends on finding a suitably skilled agent to fill the role of mercenary analyst. If the pattern succeeds, the new context defines a project whose progress can be reviewed (the pattern Review the Architecture) and monitored by community experts outside the project.
>
> ***Rationale***: Borland's Quattro Pro for Windows; many AT&T projects (a joint venture based in New Jersey, a formative organization in switching support, and others). It is difficult to find people with the skills to fill this role.
>
> Rybczynski says:
>
> > Here is another liability: beautiful drawings can become ends in themselves. Often, if the drawing deceives, it is not only the viewer who is enchanted but also the maker, who is the victim of his own artifice. Alberti understood this danger and pointed out that architects should not try to imitate painters and produce lifelike drawings. The purpose of architectural drawings, according to him, was merely to illustrate the relationship of the various parts... Alberti understood, as many architects of today do not, that the rules of drawing and the rules of building are not one and the same, and mastery of the former does not ensure success in the latter. (Rybczynski, 1989: p. 121)
>
> (adapted from Coplien, 1995a) (DeBruler, 1995)

Here is a training pattern from a delightful pattern language by Dana Anthony of Knowledge Systems Corporation, Inc.:

> **Chicken & Egg**
>
> Other possible names for the pattern include Need to Know, Simplified Mutual Prerequisites, Illusion of Understanding.
>
> . . .

Problem: Two concepts are each a prerequisite of the other. It's a "chicken and egg" situation: a student who doesn't know A won't understand B; but a student who doesn't know B won't understand A.

Constraints and Forces: You could just explain one concept and then the other, but an the halfway point, everyone would be confused. Many people, if confused, stop trying; this invalidates a "just go ahead" approach. You could just simplify each concept to the point of incorrectness, just for the sake of explaining the other one. But many people object to being lied to, even for their own good. This invalidates the "Santa Claus and Easter Bunny" approach.

Solution: Give the students the illusion of understanding, by explaining each of A and B very superficially, but essentially correctly. Iterate your explanations over and over, each time going into more detail. Be sure to maintain the illusion of understanding an each step.

Related Patterns: Pattern 3: Mix New and Old is related. As you iterate through a chicken-and-egg pair of topics, mix new material on each topic with a review of material already covered. Also, vary the "learning style" each time through. (Anthony, 1996)

Patterns may be useful in any problem-solving domain with a legacy of experience. Narrowly defined domains, such as those whose structure can easily be captured in frameworks, lend themselves to integrated pattern languages (see Section 2).

4. Classifying Patterns

The Western world view has been strongly shaped by Cartesian philosophy and its hierarchical models (in fact, this goes all the way back to Plato). Hierarchies are one of the most predominant structuring and abstraction techniques in Western thought. We see this in the procedural hierarchies of structured design and the inheritance hierarchies of basic object-oriented design.

Early pattern category models fell naturally into this hierarchical heritage. In Section 4.1, we look at early software pattern categories. As we understand patterns better, we find simple hierarchies unsatisfying. In Section 4.2, we look to Alexander for other pattern organization models.

4.1. Three Pattern Levels

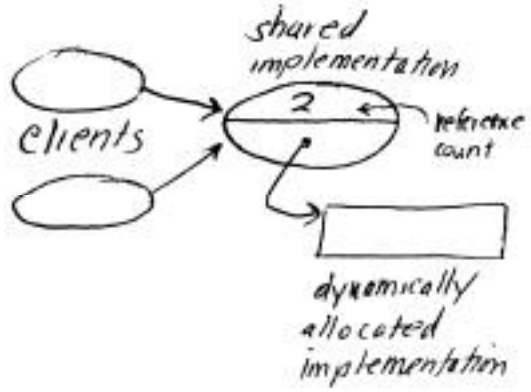
The early pattern community has converged on a layered schema of pattern categories. The layering differentiates levels of abstraction, with *idioms* at the bottom, *design patterns* in the middle, and *frameworks* at the top. This section describes each of those categories.

4.1.1. Idioms

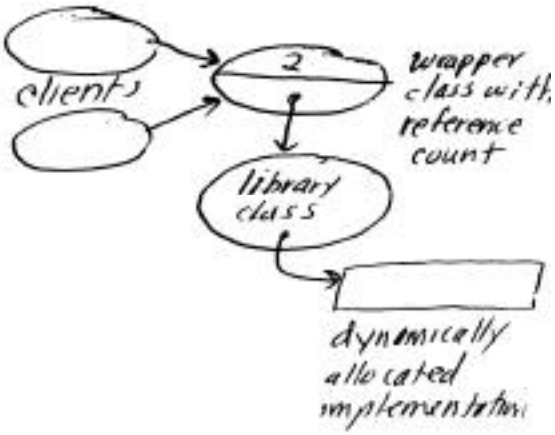
Idioms are low-level patterns that depend on a specific implementation technology such as a programming language. Idioms are among the earliest published software patterns, having emerged in late 1991 (Coplien, 1992), although early idioms were not in pattern form.

Here is an idiom cast in pattern form by the writer based on input from Andrew Koenig and Bjarne Stroustrup, who relate broad experience with the pattern:

> Pattern: Detached Counted Handle/Body
>
> …many C++ programs express types whose implementations use dynamically allocated memory. Programmers often create such types, and put them in libraries without adding the machinery to make these types as well-behaved as built-in types. The standard solution, "Counted Body Pattern," embeds a reference count in a shared implementation that is managed by a handle class:



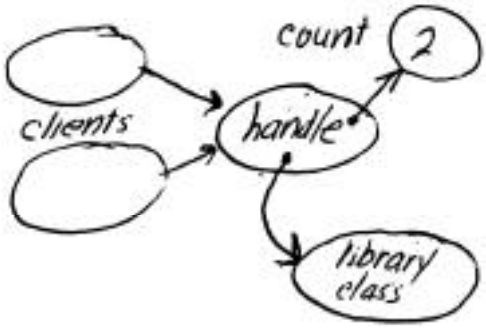
However, we may not add a reference count to a library abstraction, since we only have object code and a header file. We could solve this with an added level of indirection,



but that adds a cycle to each reference, and may be too expensive.

Therefore

Associate both a shared count and a separate shared body with each instance of a common handle abstraction:



Now we can access the body with a single level of indirection, while still using only a single indirection for the count.

Handles are slightly more expensive to copy than in "Counted Body," memory fragmentation may increase, and initial construction overhead is higher because we are allocating multiple blocks.

4.1.2. Design Patterns

Bridge (page 15) is a *design pattern*. Design patterns made their debut in the landmark book by Gamma et al. (1995). Design patterns are one level broader in scope than idioms. Their problem, forces, and solution are language independent, so they stand as general design practices for common classes of software problems.

The design patterns of Gamma et al. (1995) capture the good practices of object-oriented design independent of a particular programming language. They are microarchitectures: structures larger than objects but not large enough to be system-level organizing principles.

4.1.3. Framework Patterns

We find patterns at the system level as well. There is a close relationship between patterns and frameworks. A framework is a partially completed body of code designed to be extended for a particular application. Most frameworks build on system-level patterns that tie together sys-

tem parts and mechanisms. The pattern form is a good way to document a framework and how to extend it.

As an example of a framework-level pattern, consider the Streams framework of Stephen Edwards:

> **Problem**: This pattern allows designers to concentrate on the data flow of a complex piece of software without concern for the techniques individual components will use to distribute the computational burden...
>
> **Context**: The pattern applies to most imperative languages... Streams are most effective when the architecture of a software subsystem is best captured by highlighting the data flow within it. Thus, streams work naturally with pipe and filter-style conceptual models of program operation...
>
> . . .
>
> **Forces**: ...
>
> - The communications mechanisms that will be used between subsystem components can have a major impact on both maintainability and adaptability...
> - Different components within the subsystem may choose very different processing strategies...
> - Achieving composability of independent or semi-independent components is critical for maintainability and adaptability...
>
> **Solution**: Model each component in the subsystem as a stream of data objects... (Edwards, 1995)

Another example is Wolf and Liu's Client/Server framework pattern. This is a large umbrella pattern fleshed out by other patterns in the pattern language:

> **A Client/Server Framework**
>
> **Problem**
>
> An object-oriented client/server framework for communicating with a legacy host system, reduced to its most elemental terms, must let its user locate objects of interest, examine them, and change (or create) them.
>
> **Solution**
>
> The client/server framework pattern, then, is an aggregation of *searching*, *viewing*, and *updating/creating* patterns. A successful framework depends on the quality with which these three patterns are implemented, which in turn depends recursively on the quality with which their constituent patterns are implemented. (Wolf & Liu, 1995)

This pattern builds on other patterns that can be found in the same reference.

4.1.4. Problems with the Three Levels

Splitting patterns into three levels of abstraction—idiom, design pattern, and framework—is arbitrary in the sense that abstraction falls along a continuum without clear boundaries. This makes classification subjective.

But there is a deeper issue as well. Many patterns transcend all three levels of architecture. Consider Model-View-Controller, which started as an idiomatic artifact of the Smalltalk culture. As it became more broadly understood, it became a design staple for object-oriented user interface design. Buschmann et al. (1995: p. 334) use it as a framework-level pattern because it is the primary structuring principle at the highest levels of the system. Patterns this broad are difficult to categorize according to this three-level abstraction taxonomy.

The three-level organizing scheme has serious limitations and ambiguities. There are other good organizing principles that we can build on as we organize and index pattern catalogues and pattern languages. Some hark back to Alexanderian roots.

4.2. Alexanderian Scaling

Alexander (1979; p 325) talks about building a general pattern language from specific ones. He is fascinated by similar patterns in multiple pattern languages. For example, he notes "that ENTRANCE TRANSITION is part of the language for the garden, and also part of the language for the house." We see related patterns in distinct design domains too: the CHECKS pattern language (page 41) recurs in the IEEE floating point standard as "Not a Number." Such similarities point to more general patterns at a higher level.

We find this same kind of layering in architectural patterns. Consider the following two patterns from telecommunication systems. Riding Over Transients is a broad pattern that "contains" or "builds on" the "smaller" pattern, Leaky Bucket Counter:

> **Pattern**: Riding Over Transients
>
> **Alias**: Make sure problem really exists
>
> **Problem**: How do you know whether a problem will work itself out or not?
>
> **Context**: A fault-tolerant application where some errors, overload conditions, etc. may be transient. The system can escalate through recovery strategies, taking more drastic action an each step. A typical example is a fault tolerant telecommunication system using static traffic engineering, where you want to check for overload or transient faults.
>
> **Forces**:
>
> You want to catch faults and problems.
>
> There is no sense in wasting time or reducing level of service trying to solve a problem that will go away by itself
>
> Many problems work themselves out, given time.
>
> **Solution**: Don't react immediately to detected conditions. Make sure the condition really exists by checking several times, or use Leaky Bucket Counters to detect a critical number of occurrences in a specific time interval. For example: by averaging over time or just by waiting a while, give transient faults a chance to pass.
>
> **Resulting Context:** Errors can be resolved with truly minimal effort, because the effort is expended only if the problem really exists. It allows the system to roll through problems without its users noticing, or without bothering the machine operator to intervene (as in the pattern Minimize Human Intervention).
>
> **Rationale**: This pattern detects "temporally dense" events. Think of the events as spikes on a time line. If a small number of spikes (specified by a threshold) occur together (where "together" is specified by the interval), then the error is a transient. Used by Leaky Bucket Counters, Five Minutes of No Escalation Messages, and many others.
>
> Author: James 0. Coplien (Adams, et al., 1996)

Here's the second pattern, Leaky Bucket Counters. It's a specialization of the more general Riding Over Transients:

> **Name**: Leaky bucket counters
>
> **Problem**: How do you deal with transient faults?
>
> **Context**: Fault-tolerant system software that must deal with failure events. Failures are tied to episode counts and frequencies.
>
> One example from 1A/1B processor systems in AT&T telecommunication products: As memory words (dynamic RAM) got weak, the memory module would generate a parity error trap. Examples include both 1A processor dynamic RAM and 1B processor static RAM.
>
> **Forces**: You want a hardware module to exhibit hard failures before taking drastic action. Some failures come from the environment, and should not be blamed on the device.

Solution: A failure group has a counter that is initialized to a predetermined value when the group is initialized. The counter is decremented for each fault or event (usually faults) and incremented on a periodic basis; however, the count is never incremented beyond its initial value. There are different initial values and different leak rates for different subsystems: for example, it is a half-hour for the 1A memory (store) subsystem. The strategy for lA dynamic RAM specifies that the first failure in a store (within the timing window) causes the store to be taken out of service, diagnosed, and then automatically restored to service. On the second, third, and fourth failure (within the window) you just leave it in service. For the fifth episode within the timing window, take the unit out of service, diagnose it and leave it out.

If the episode transcends the interval, it's not transient: the leak rate is faster than the refill rate, and the pattern indicates an error condition. If the burst is more intense than expected (it exceeds the error threshold) then it's unusual behavior not associated with a transient burst, and the pattern indicates an error condition.

Resulting Context: A system where errors are isolated and handled (by taking devices out of service), but where transient errors (e.g., room humidity) don't cause unnecessary out of service action.

Rationale: The history is instructive: In old call stores (lA memories that contained dynamic data), why did we collect data? For old call snores, the field replaceable unit (FRU) was a circuit pack, while the failure group was a store comprising 12 or 13 packs. We needed to determine which pack is bad. Memory may be spread across 7 circuit packs; the transient bin was only one bin, not enough to isolate the failure. By recording data from four events, we were better able to pinpoint (90% accuracy) which pack was bad, so the machine operator didn't have to change 7 packs.

Why go five failures before taking a unit out of service? By collecting failure data on the second, third, and fourth time, you are making sure you know the characteristics of the error, and are reducing the uncertainty about the FRU. By the fifth time, you know it's sick, and need to take it out of service.

Periodically increasing the count on the snore creates a sliding time window. The resource is considered sane when the counter (re-) attains its initialized value. Humidity, heat, and other environmental problems cause transient errors which should be treated differently (i.e., pulling the card does no good).

See, for example, Fool Me Once, which uses simple leaky bucket counters.

This is a special case of the pattern Riding Over Transients.

Strategy alluded to in p. 2003-4 OF BSTJ XLIII 5(10), Sept. 1964.

Author: Robert Gamoke (Adams, et al., 1996)

4.3. Other Scaling Approaches

Pattern authors have used other techniques to organize patterns. One common approach has been to use umbrella patterns that tie multiple smaller patterns together. The smaller patterns may be disjointed, with the umbrella patterns as the sole locus of the relationship between them. This is how patterns are organized within a pattern language. The Client/Server pattern of Wolf and Liu (see page 25) heads such a pattern language.

4.4. Anti-Patterns

Anti-patterns are literature written in pattern form to encode practices that don't work or that are destructive. Anti-patterns were independently pioneered by Sam Adams,

Andrew Koenig (Koenig, 1995), and the writer. Many anti-patterns document the rationalizations used by inexpert decision makers in the Forces section. Consider this organizational anti-pattern, Egalitarian Compensation:

> **Name**: Egalitarian Compensation
>
> **Problem**: Providing appropriate motivation for success.
>
> **Context**: A community of developers meeting night schedules in a high-payoff market.
>
> **Forces**: Disparate rewards motivate those who receive them, but may frustrate their peers. You want to encourage team cohesion, build team identity, and in general encourage team behavior.
>
> **Supposed solution**: The entire team (social unit) should receive comparable rewards, to avoid demotivating individuals who might assess their value by their salary relative to their peers.
>
> **Resulting Context**: An organization where people feel accepted as peers because, financially, they are peers. However, leaders will still emerge and there will still be an inequitable distribution of work; that distribution of work is no longer commensurate with compensation.
>
> People figure this out, and lose one of their motivations to excel. The pattern has the opposite effect of encouraging behavior where people over-extend themselves.
>
> **Rationale**: Note that this differs from Compensate Success (Coplien, 1995a) in only one detail: that outstanding contributors don't receive exceptional rewards. High rewards to some individuals may still demotivate their peers, but rewarding on a team basis helps remove the "personal" aspect of this problem, and helps establish the mechanism as a motivator, in addition to being just an after-the-fact soother.
>
> **Author**: James 0. Coplien (originally posted to the WikiWikiWeb, http://c2.com/cgi/wiki ?Egalitarian-Compensation).

This pattern might be viewed as the "shadow" of Compensate Success (Coplien, 1995a), but not every pattern has a corresponding anti-pattern.

Anti-patterns don't provide a resolution of forces as patterns do, and they are dangerous as teaching tools: good pedagogy builds on positive examples that students can remember, rather than negative examples. Anti-patterns might be good diagnostic tools to understand system problems.

4.5. Meta-Patterns

Wolfgang Pree's book (Pree, 1995) introduced the term *meta-pattern* to the software pattern community. It may be a misnomer: there isn't much "meta" about the patterns, although they are building blocks from which other patterns can be built. In particular, one can view Pree's meta-patterns as generalizations of key structural (implementation) aspects of many of the GOF patterns (Gamma et al., 1995).

One interesting aspect of Pree's work is the concept of *hot spots*, which are the locations in a framework where parameterization takes place. Pree sees hot spots as crucial places to apply patterns.

For a broad and thought-provoking look at meta-patterns in general, see Volk (1995).

4.5. Patterns and Strategies

The book *Object Models: Strategies, Patterns and Applications* by Coad, North, and Mayfield (1995) uses the term pattern to refer to object-oriented design templates. How-

ever, the book shows almost no ties to the mainstream pattern discipline, and the patterns lack effective rationales and criteria for application that are central to Alexanderian patterns.

For a review of this work that further explores its relationship to patterns, see Berczuk (1996a).

5. Pattern Pragmatics

Much of this paper focuses on principles, philosophies, and values behind patterns. Before we move on to the deeper subject of generativity and the pattern value system, it might be good to spend some time appealing to the technical management community. What should you expect patterns to do for you? And what should you expect them not to do for you?

5.1. The Goal of Patterns

Cost, customer satisfaction, productivity, and development interval reduction are among the holy grails of software development. Patterns contribute indirectly to many of these goals:

- **Productivity**. By providing domain expertise, patterns short-circuit the *discovery interval* for many important design structures. "Discovery" includes a designer's activities to find out how the current system works as a basis for maintenance changes.

  More importantly, patterns avoid rework that comes from inexpert design decisions. As an example, programmers who don't understand idioms like the Counted Body Idiom (page 15) either will spend a long time converging on the solution, or will employ solutions that are less maintainable or just plain wrong.

- **Development Interval**. Many software patterns are a form of design-level reuse. Patterns can reduce the amount of time required to build solution structures because they allow designers to use design chunks that are larger than functions or objects.

  Patterns also provide road maps to the structure of existing systems, making it easier for the inexpert designer to understand and navigate existing software. This can reduce discovery costs. Our studies an AT&T suggest that as much as half of software development effort can be attributed to discovery.

- **Cost**. Cost reduction follows in a straightforward way from development interval reduction.

- **Customer Satisfaction**. Customer satisfaction is largely a result of the other factors.

Recently, a large project at AT&T documented its key architectural patterns as part of the project architecture documentation. These patterns proved to have strategic value to a successor project, both because they educated a new development community about *what* the new system needed to do and about *how* to achieve the required functionality.

Of course, your mileage will vary. Patterns alone promise none of these benefits. They are a tool that amplifies human expertise and the power of applicable technologies, good management practices, and good business practices and opportunities.

5.2. What Patterns Can't Do

- Since patterns capture experience, some important patterns of new business domains won't be there to support development of the first system.

We can argue that many important system patterns transcend domains and that new business domains can build on patterns from other domains. This is true to a degree, particularly for general patterns such as object-oriented design patterns, distributed processing patterns, client/server patterns, and the like. But each new domain begs new patterns that won't be thereat the outset. Such patterns will come only with experience.

- Patterns guide humans, not machines. They will not generate code; they do not live inside CASE tools. They are literature that aids human decision-making processes. Patterns should not, cannot, and will not replace the human programmer.

- Although patterns leverage the knowledge of acknowledged experts, they will not turn everyday workers into experts. Patterns keep everyday designers from making inexpert errors and invite them to consider new principles. Patterns themselves cannon inculcate the intuitive sense of aesthetics we find in truly great designers; these are intuitive, almost instinctive. Perhaps long-term experience with patterns can help novices evolve into experts, as any long-term experience can lead the right people into true expertise.

5.3. A Pattern Program

Patterns are not a design method, but they can be used as an adjunct to any design method. Patterns don't require a major shift in organization or process, but they do depend on a supportive culture. This culture includes pattern training, pattern mining, pattern publication, and pattern application.

5.3.1. Pattern Training

We run a two-day workshop to help people become proficient with patterns. We spend a bin of time relating the history: the place of Alexander as inspiration and as the source of much of the terminology. We describe what problem patterns are trying to solve: to capture corporate intellectual assets, particularly those of architecture, in a body of literature. We describe the basics of the pattern form: problem, context, forces, solution, resulting context.

To teach designers about architectural patterns, we often need to relate architectural basics. We emphasize that architecture is about relationships between parts, not just about interfaces and cohesion. We emphasize that architecture serves a human need: to free development teams to work as independently as possible, decoupled from other groups along architectural force lines.

But the key aspect of pattern training comes when students write patterns. We ask them to think of one thing they know that is important but they think isn't widely appreciated. We ask them to struggle with the forces, and to put their gem into pattern form. This exercise serves several purposes. First, it underscores the importance of forces to the student. Second, it gives the student a very up-close tour of patterns, so they can read them more instructively. Third, it helps the student realize that everyone has something important to contribute; this perspective helps combat the notion that patterns must be esoteric and pompous. This exercise helps underscore important aspects of the pattern value system.

5.3.2. Pattern Mining

We use common industry patterns like the GOF patterns, but many of our key patterns aren't object oriented an all. Every business thrives on the patterns it's refined over decades of evolution. They are the gems you want to seek out and document.

A pattern-mining exercise should focus on the key intellectual (or business) assets of an enterprise. Ask the enterprise: what are the key technical strategies that have made the company great (quick product turnaround, availability, consistent performance, etc.)? Approach experts in these areas and ask them about the particulars; many of their answers will be patterns, and many others can easily be transformed into patterns. (I know this sounds presumptuous, but try it—it has worked for many organizations using patterns for the first time). The knowledge elicitation exercise is similar to that used by knowledge engineers, and it is fraught with the same problems. We overcome these problems by employing "junior experts" as "interpreters" in the interviews—they learn stuff and have fun, too!

I held each interview session to one-and-a-half hours, and transcribed the patterns into HTML (Hyper-Text Markup Language, the lingua franca of the World Wide Web) myself without any fancy tools. The pattern acquisition rate was high for us, perhaps because of luck or because of the experts' personalities.

It helped if the interviewees had had some exposure to patterns, but we had good experiences with domain experts 'who had never heard of patterns.

One can take a more bottom-line—oriented approach to pattern mining. First, identify the areas in which you know your organization is lacking. Find someone who knows how to handle those areas well and mine patterns from them. I remember working on a project where only a single project manager consistently met his schedules, and we mined his managerial patterns to see what allowed him to succeed while others failed.

It's crucial to mine patterns from people, organizations, or products with long, proven track records.

5.3.3. Pattern Publication

Patterns enjoy publication both via the World Wide Web, usually as HTML documents, and in the conventional press. There isn't too much remarkable about pattern publication except that hypermedia seem to be particularly valuable as a navigation aid, especially for pattern languages.

More important than the format of publication is the review process leading up to publication. Good patterns are timeless literature. By "timeless," we mean literature with broad appeal that is easily understood, and which captures timeless design wisdom. It goes without saying that very little timeless literature has emerged from the engineering community.

One important aspect of the pattern publication process is the pattern review. A good review can help ensure that the pattern is readable, that it hits its audience, and that it does its job. Patterns are literature, and we believe they deserve more than the dry design reviews we typically afford other software documents. Richard Gabriel, himself a poet (in addition to being one of the inventors of CLOS), naught us how poetry is reviewed at poetry conferences. We adapted that approach and call these sessions *writers workshops.*

The participants in a workshop are all pattern authors. The author reads a selection from the pattern, and then sins down and becomes "invisible" until the very end of the session. One of the reviewers summarizes the pattern. There is then open discussion on why the pattern works and what's good about it. Next, reviewers can make suggestions on how to improve the pattern: content, form, usage—anything is fair game. Finally, the author can ask for clarification. We found that each session took about an hour.

There are many pattern forms (Section 1.3); which form should you use? Experiment or choose a form you're comfortable with. All are equally effective under suitable conditions.

5.3.4. Pattern Application

There is little magic to pattern application. Once patterns are in place, they can be used as reference material by the design community. Rather than an oracle for fixing problems that arise, it is better to use patterns as a domain tutorial for designers. Designers should skim all catalogued patterns for their domains at least once. As design problems arise, designers can return to the original patterns for detailed guidance.

Broadened exposure and experience helps to refine patterns over time, enlarging their context and illuminating their forces and rationales. A good pattern is living literature that matures with age.

5.3.5. Where to Go from Here

This has been a shorn section, partly because it's difficult to codify the correct steps or proven program for success. Each culture should tune patterns to its needs. Much of the process must be learned from experience. Learning from others' experiences may also help; for more insights, see the experience paper by Beck et al. (1996).

6. Generativity

In many problem-solving strategies, we try to attack problems directly. In doing so, we often attack only symptoms, leaving the underlying problem unresolved. Alexander understood that good solutions to architectural problems go at least one level deeper. The structures of a pattern are not themselves solutions, but they *generate* solutions. Patterns that work this way are called *generative patterns*. A generative pattern is a means of letting the problem resolve itself over time, just as a flower unfolds from its seed:

> *9. This quality in buildings and in towns cannot be made, but only generated indirectly by the ordinary actions of the people, just as a flower cannot be made, but only generated from the seed* (Alexander, 1979. p. xi)

And later:

> An ordinary language like English is a system which allows us to create an infinite variety of one dimensional combinations of words, called sentences. . . . A pattern language is a system which allows its users to create an infinite variety of those three dimensional combinations of patterns which we call buildings, gardens, towns.
>
> . . .
>
> Thus, as in the case of natural languages, the pattern language is generative. It not only tells us the rules of arrangement, but shows us how to construct arrangements—as many as we want—which satisfy the rules. (Alexander, 1979: pp. 185—186)

Like many other facets of Alexander's philosophy, this philosophy can be traced back to Eastern schools of thought (Lao Tsu principles of nonaction, part 3). This generativity is an important aspect of the Quality Alexander seeks. It is an elusive quality, so elusive he calls it "the quality without a name"; we'll revisit that in Section 7.1, but we need not turn to esoteric sources for insights on the importance of generativity in problem-solving; other contemporary sources will do. In Sengé, we find:

> What, exactly, does it mean to say that structures generate particular patterns of behavior? (Sengé, 1990: p. 45)
>
> ... a fundamental characteristic of complex human systems ... [is that] "cause" and "effect" are not close in time and space. By "effects," I mean the obvious symptoms that indicate that there are problems— drug abuse, unemployment, starving children, falling orders, and sagging profits. By "cause" I mean the interaction of the underlying system that is most responsible for generating the

> symptoms, and which, if recognized, could lead to changes producing lasting improvement. Why is this a problem? Because most of us assume they *are*—most of us assume, most of the time, that cause and effect *are* close in time and space. (Sengé, 1990: p. 63)

At this writing, few published software patterns exhibit generativity. Here is a great generative pattern from a domain outside software:

> **Name**: Hands In View
>
> **Problem**: The skier fails to commit downhill on steeps and bumps, resulting in slides, backward falls, and "yard sales."
>
> **Context**: In order to explore the entire mountain environment, a skier must be comfortable and adaptable to any terrain and rapid terrain change. To take advantage of this pattern the skier should be skiing at a level at which parallel turns can be linked consistently.
>
> **Forces**:
>
> Fear of falling is the most basic of all responses
>
> Reliance on equipment is essential
>
> Continuous movement is essential
>
> Fatigue can be a factor in long descents
>
> Commitment downhill over skis is essential for skis to function as designed
>
> **Solution**: Concentrate on keeping the hands in view. Bring then into sight immediately after each pole plant and turn.
>
> **Resulting Context**: Keeping the hands in view changes the alignment of the body from sitting timidly back and allowing the edges to skid out from under the skier. Thus, keeping the hands in view pulls the body forward and thus downhill, bringing the skier's weight over the downhill ski, forcing the edge to bite and turn.
>
> **Rationale**: As steepness increases, the natural tendency of any sane person is to sit back against the hill and retain the perpendicularity the inner ear prefers. Unfortunately, skis must be weighted to perform as designed, the weight causing flex, which in turn pushes the edges into the snow in an arc, making a turn. Therefore it is essential to "throw" oneself down the mountain and over the skis, depending on them to "catch" the fall as they bite into the snow to turn underneath the perpetually falling skier. Intellectually this can be clearly understood but fear prevents execution. Concentrating on something as simple and indirect as "look an your hands" causes the desired behavior without directly confronting the fear. This is directly analogous to what occurs when an individual walks: the weight is thrown forward in a fall, with the consequent forward thrust of the leg to catch this fall, repeated for left and right sides in a continuous tension and release of yielding to gravity in order to defy it.
>
> **Author**: Don Olson 95/07/07
>
> **Originator**: Anonymous ski instructor somewhere in Utah. Wherever you are, thanks for providing the breakthrough to better skiing for the author. (Personal correspondence with Don Olson, July 1995)

Why is generativity important? First, as Sengé says, most real problems go deeper than their surface symptoms, and we need to address most interesting problems with emergent behavior. Second, a good pattern is the fruit of hard work and intense review and refinement. Simple problems can be addressed through simple rules, since the solutions are more direct or "obvious" than we find in generative solutions. The pattern form excels an engaging the reader in generative solutions: to understand the principles and

values of lasting solutions and long-term emergent behavior. Good patterns go beyond the quick fix.

Pattern languages as a whole may exhibit a "gestalt" generativity. Each of the patterns of the CHECKS language (page 41) solves a problem, but the language as a whole solves a much broader software engineering problem than is addressed by any pattern alone. This "surprise solution" is a form of generativity. Caterpillar's Fate (Kerth, 1995) has this same property.

7. The Pattern Value System

Software patterns usually have a strong technical component, which hopefully has come through in the patterns presented here, driven by the principles we've discussed. These principles include the importance of design and architecture, particularly as it extends beyond modular partitioning, principles of form and organization, and deep principles such as generativity.

Many of these principles are driven by an even deeper value system. We explore that value system here in Section 7. This is a long section for two reasons. First, this material is little understood or socialized in the software community, and this forum provides an opportunity to raise industry consciousness about patterns. Second, this material is particularly important. These values support principles, and these principles support the goals of software development or of most any human endeavor: human comfort, what Alexander calls being "fully alive."

Many of these values come from Alexander. Though many of Alexander's building architecture principles don't translate into software, the essential human values do. Some of these values come from the early software patterns community, and are less nightly linked to Alexander. They are nowhere codified today that I know of.

However, one finds these values extolled an pattern conferences and in much of the pattern literature; they are part of the emerging pattern culture.

Do all pattern practitioners adhere to all features of this value system? Certainly not. Many pattern users draw on the existing literature simply as a design reference. But these values are a visible feature of the culture of pattern workshops and conferences and of other gatherings of authors, editors, pattern reviewers, and researchers in the mainstream pattern discipline.

7.1. The Quality Without a Name

Alexander strives for the "quality without a name" in the patterns he captures and in the rooms, buildings, and towns they engender. We seek a similar quality in the software we build with patterns.

What is this nameless quality? Most coders have had the pleasure of knowing it, when they build a particularly *satisfying* module or system, code that just "feels right." Of course, to define it or name it would miss the point. However, recurring themes in the pattern literature point out aspects of what such a Quality might be.

One theme, close to Alexander's goals of architecture, is to serve human needs. We too often overlook that all software serves a human need; the pattern form, particularly in the forces and resulting context, is an opportunity to draw attention to these needs. We take up this issue in Section 7.8 on aesthetics, but it goes beyond aesthetics to the broader question of human comfort. We will take up this topic again in Section 7.7.

Generativity (Section 6; in particular, note the quote from Alexander) is an important part of the Quality, since it reflects a deep understanding of the problem, and a deeply rooted solution.

Alexander notes that the Quality is “slightly bitter” in its reminding us that nothing is permanent. Alexander thought nothing of using the shade of a neighboring tree to orient a house for light exposure, although one might think that the timeless house shouldn’t defer to something as short-lived as a tree. To knowingly recognize and build to today’s environment, knowing that the environment will change, helps put evolution into perspective. This evolution and consciousness of time are central to the Quality.

The quality without a name is certainly subjective, and it may mean something different to each designer. In the rest of this section, we explore other values that feed the edges of this quality, values that have taken root in the pattern culture.

7.2. Real Stuff

Patterns are about “real stuff.” They capture proven practice, rather than postulates, theories, or detached models of techniques that might work.

Kent Beck relates a story from the 1987 workshop where some of the earliest discussions of software patterns took place (see Beck, 1988). His patterns offered design principles that built on repeated success in application, principles that could be even more broadly applied if people knew about them. Another workshop attendee gave a talk based on the premise that even though there was not broad experience with a particular technique, all would be rosy if only ordinary software engineers would start using a particular set of techniques that had been developing in academic research an that time. We want to celebrate what people do that works, not tell them to do something else on the basis only of our intuition, arguments, and aspirations.

A good pattern draws on real examples in its rationale section; some forms have their own examples section (see Section 1.3). As a rule of thumb, a good pattern should have three examples that show three insightfully different implementations.

In this regard, the patterns community takes up the quote attributed to W. Edsgar Dijkstra: “Premature abstraction is the root of all evil.”

We focus on real stuff in the solution section of a pattern. A solution should tell the reader to do something specific and concrete.

Yet, the concreteness of real stuff isn’t inconsistent with abstraction. *Abstract* does not mean vague. An abstraction, while lacking detail (by definition), can be crisp, sharp, and instructive. We can draw an analogy from software design. Though the abstract interface of a String class distances the user from the implementation, it is all that is visible to the user. Together with documentation that explains its use, the abstract interface is sufficient for effective use. A good pattern should be the same way: sufficiently abstract that it could be implemented a million times without ever being the same thing twice, yet specific enough to tell the user what to do.

Yet Gabriel also admonishes us to look beyond the abstraction to what lies inside; that is where many important patterns lie:

> But what of geometry? Alexander always goes back to this. And one of his key questions is this: What is it that dictates geometry possessing the quality without a name—what is that thing that is falsely called “simplicity”?
>
> What corresponds to geometry for us?
>
> I think it is the code itself. Many talk about the need for excellent interfaces and the benefits of separating interface form implementation so that the implementation may vary. But few people talk seri-

> ously about the quality of the code itself. In fact, most theorists are eager to lump it into the category of things best not discussed, something to be hidden from view so that it can be changed in private. But think of Alexander's remarks above: the quality comes in nearly equal part from the artistry and creativity of the builder who is the one whose hands most directly form the geometry that gives the building its quality and character. Isn't the builder the coder? And isn't the old-style software methodology to put design in the hands of analysts and designers and to put coding in the hands of lowly coders, sometimes offshore coders who can be paid the lowest wages to do the least important work? (Gabriel. 1996: p. 68)

7.3. Constraints Are Liberating

The pattern form constrains the pattern writer. Removing form from the problem-solvers' consideration frees them to focus on other things. We feel that the pattern style is better than unstructured natural language because of the guidance it offers, and that it is also better than formats like SGML. SGML forms don't understand the semantics and relationships of pattern sections, so they don't offer enough constraints. The main benefit SGML forms—and in particular HTML, an SGML derivative—is their wide acceptance as a portable publication format.

The finished pattern constrains the developer. A good pattern helps the developer focus on the problem by drawing out important forces. This can help keep the developer from unnecessarily exploring blind alleys. Developers still must apply their design insights and experience—a good pattern leaves considerable leeway for creative adaptation.

Alexander himself says:

> The rules of English make you creative because they save you from having to bother with meaningless combinations of words. . . . A pattern language does the same. (Alexander, 1979: pp. 206—207)

7.4. Participative Architecture

Alexander believed that people should participate in the design of their dwellings instead of deferring to a professional architect. Architecture meets deeply human needs, and it is more trouble for an architect to assimilate the needs of an individual—both those owing to personal preferences and those rising from social context—that it is for individuals to derive the design themselves.

This same spirit is at the foundation of the software pattern community. This was one of the early strategies that Kent Beck and Ward Cunningham first employed in patterns for human interface design: to let the users of the system design its human interface, as Alexander said the occupants of a building should design it.

The writer talks about this principle in one of his organizational patterns, Architect Also Implements:

> Architect Also Implements...
>
> **Problem**: Preserving the architectural vision through to implementation
>
> **Context**: An organization of *Developers* that needs strategic technical direction.
>
> **Forces**: Totalitarian control is viewed by most development teams as a Draconian measure. The right information must flow through the right roles.
>
> **Solution**: Beyond advising and communicating with *Developers*, *Architects* should also participate in implementation.

Resulting Context: A development organization that perceives buy-in from the guiding architects, and that can directly avail itself of architectural expertise.

Design Rationale: The importance of making this pattern explicit arose recently in a project I work with. The architecture team was being assembled across wide geographic boundaries with narrow communication bandwidth between them. Though general architectural responsibilities were identified and the roles were staffed, one group had expectations that architects would also implement code; the other did not.

One manager suggests that, on some projects, architects should focus only on the implementation of a common infrastructure, and that the implementation of non-core code should be left solely to the *Developer* role.

Rybczynski tells us, "It would be convenient if architecture could be defined as any building designed by an architect. But who is an architect? Although the Académie Royale d'Architecture in Paris was founded in 1671, formal architectural schooling did not appear until the nineteenth century. The famous École des Beaux-Arts was founded in 1816; the first English-language school, in London, in 1847; and the first North American university program, at MIT, was established in 1868. Despite the existence of professional schools, for a long time the relationship between schooling and practice remained ambiguous. It is still possible to become an architect without a university degree, and in some countries, such as Switzerland, trained architects have no legal monopoly over construction. This is hardly surprising. For centuries, the difference between master masons, journeymen builders, joiners, dilettantes, gifted amateurs, and architects has been ill defined. The great Renaissance buildings, for example, were designed by a variety of non-architects. Brunellesehi was trained as a goldsmith; Michelango as a sculptor, Leonardo da Vinci as a painter, and Alberti as a lawyer; only Bramante, who was also a painter, had formally studied building. These men are termed architects because, among other things, they created architecture—a tautology that explains nothing." — Witold Rybczynski, *The Most Beautiful House in the World*, page 9.

Vitruvius notes: ". . . [A]rchitects who have aimed at acquiring manual skill without scholarship have never been able to reach a position of authority to correspond to their pains, while those who relied only upon theories and scholarship were obviously hunting the shadow, not the substance. But those who have a thorough knowledge of both, like men armed at all points, have the sooner attained their object and carried authority with them." (Vitruvius, *The Ten Books of Architecture*, translated by Morris Morgan. New York: Dover Publications, 1960: p. 5) (Coplien, 1995a)

Just as this pattern draws on classic architecture to motivate software architects to "keep their fingers dirty," so the pattern discipline calls on software architects to stay grounded in the implementation of their designs. Too many software methods suffer from "over-the-wall" design transfer, which almost always assures that the original design is lost.

When we capture design and architecture patterns, it's important to capture them in a way that day-to-day programmers can use them. This is consistent with the pattern agendas to capture *real stuff* (Section 7.2) and to preserve the dignity of programmers (Section 7.5).

7.5. Dignity for Programmers

Because patterns are about real stuff, they speak to the daily concerns of the programmer. This draws attention to the unheralded elegance and power of the techniques that shape the customer-deliverable artifact. Such an emphasis

breaks with the disproportionate respect our discipline has accorded architecture and formal methods. Most software cultures honor the lead architect and the methodologist, but keep those who cast the product in its final form low on the social ladder. These values seem to mirror the industrial age and its social structure of factory assembly lines, where programmers adopt the stature of the assembly line worker. Far too many cultures view their programmers as little more than unskilled workers.

In the pattern community, we seek to celebrate the excellence of designers and programmers. Their contributions shape end products to as large or greater extent than do those of the architects.

By the same token, we believe that architects need to earn their keep by keeping in touch with the reality of implementation. That fits well with patterns as real stuff (Section 7.2); we find this principle extolled in a process pattern language such as Architect Also Implements (see Section 7.4).

7.6. Aggressive Disregard for Originality

Many new computer science techniques and technologies distinguish themselves at the expense of their predecessors. For example, advocates of object-oriented techniques often extol them by comparing them to procedural design techniques, illustrating their superiority for solving well-known problems. Brian Foote uses the term “novelty vulture” to describe the behavior expected of leading engineers.

Anyone who has been in the industry for a long time can point to cycles of new technology, and can tell you that the expectations an the rise of a new technique always exceed what can be delivered later.

We seek to capture long-proven ideas in patterns. This breaks with the cultural norms of most R&D organizations that reward innovation, invention, and novelty. In the pattern community, we hold an aggressive “disregard for originality” (more of Brian’s phraseology).

This is not to say that novel solutions should be dismissed our of hand. Existing patterns can be applied in novel ways, such as Buschmann’s application of Model-View-Controller at the highest levels of system architecture (Buschmann & Meunier, 1995). Since today’s original ideas will be the patterns of the next decade, it’s important to keep innovative developments alive.

Patterns help reduce risk on large, new ventures that can build on the fundamentals of past, successful ventures. Good design balances patterns (in the main structure of the solution) with new techniques and methods. For example, a telecommunication system architect would be unwise to ignore the pattern language in Section 2.1. These patterns shape the overall structure and mechanisms in ways that fundamentally support business needs, but these patterns might be perfectly compatible with an object-oriented or rule-based implementation. Balancing the old patterns with the new is just an example of the risk management concerns that have always been the purview of software project managers.

This raises an important detail: patterns have no ipso facto relationship to objects. Good patterns exist in the good practices of many prior disciplines; many of the patterns in this briefing are of that nature. Such patterns are rarely incompatible with object-oriented techniques, but they wouldn’t be recognized as object-oriented techniques per se. Furthermore, patterns work well outside the realm of software method itself in such areas as organization and training; these are hardly object-oriented disciplines.

At some point, old patterns cease to work and new patterns take their place, but such changes rarely take place subtly or incrementally; they happen as fundamental paradigm shifts. Paradigm shifts are hard because peo-

ple have difficulty letting go of the patterns with which they are comfortable. But paradigm shifts are also rare: most of the time, nothing happens (Weinberg, 1988), and tried-and-true principles prevail over invention.

7.7. The Human Element

In an electronic mail exchange, Richard Gabriel summarized our concern for human issues well:

> We are trying to bring people and humanity into the software design and development process. I think this is the goal because it's the key Alexandrian idea. If you look an his patterns, each or nearly all talk about the context and forces in terms of what people need to do to live fully and to be fully alive. So far our patterns are not like this (when I get a spare moment, I will write out and send some very, very simple patterns I have that focus on people as the context and forces and which have a technical thing as their "solution"). I believe we need to get clear on this before we subject ourselves to too much togetherness with the outside world—because this position is probably considered nuts to the theoretical community we might deal with. (Gabriel, 1995)

All software serves a human need, at some level. During design, engaging these human issues provides an important perspective—perhaps the most important perspective—on the relevance of architecture and design activities. We will see this in Gabriel's pattern Simply Understood Code in the next section (page 40), which follows from the above citation. We also saw a strong human element in the telecommunications patterns of Section 2.1 on page 18. While the casual reader might think of these as architectural patterns, the careful reader will discover that many have little to do with software structure, focusing instead on human behavior, human needs, and human motivations.

Many traditional, supposedly technical principles carry deeper human forces. Frank Buschmann and Regine Meunier (Buschmann & Meunier, 1995) include cohesion and coupling in their pattern classification scheme. Coupling and cohesion, pioneered by Constantine, are time-honored principles of software design. The software itself doesn't care about its coupling and cohesion; if anything, the software performance could benefit from tighter coupling that overcomes some of the inefficiencies of abstract interfaces. But we focus on abstraction, coupling, and cohesion for the sake of the people who own and maintain the code. People can work independently to the extent that their code is decoupled from the code of other teams. Good patterns capture not only the principles of coupling and cohesion at the code level, but also the forces and resulting context at the human level. To fall short in the human area while succeeding in the technical domain is to have missed the point of a pattern and of the principles of coupling and cohesion in their own right.

Aesthetics, taken up in the next section, are an important aspect of the human element of patterns.

7.8. Aesthetics

Computer science is popularly billed as a science. Academia teaches testable skills, and managers try to manage according to measurable quantities. While it's important to teach the basics, and while a development project needs a common discipline, these factors aren't the underpinnings of software quality.

The aesthetics of the design itself are good indicators of system maintainability. A system that can't easily be understood can't easily be evolved. Good design appeals to the human aspects of development.

Patterns celebrate software as literature. The pattern community is not the first to do so; Knuth brought us literate

software (Knuth, 1995). Software development is a creative task. This means that no two people will solve the same problem exactly the same way, or that an individual will solve a problem the same way twice.

Aesthetics are important because of the human element. Aesthetics in code make code easier to understand, and more maintainable. Consider the following pattern from Richard Gabriel:

> Pattern: Simply Understood Code
>
> …at the lowest levels of a program are chunks of code. These are the places that need to be understood to confidently make changes to a program, and ultimately understanding a program thoroughly requires understanding these chunks.
>
> In many pieces of code the problem of disorientation is acute. People have no idea what each component of the code is for and they experience considerable mental stress as a result.
>
> Suppose you are writing a chunk of code that is not so complex that it requires extensive documentation or else it is not central enough that the bother of writing such documentation is worth the effort, especially if the code is clear enough on its own. How should you approach writing this code?
>
> People need to stare at code in order to understand it well enough to feel secure making changes to it. Spending time switching from window to window or scrolling up and down to see all the relevant portions of a code fragment takes attention away from understanding the code and gaining confidence to modify it.
>
> People can more readily understand things that they can read it their natural text reading order; for Western culture this is generally left to right, top to bottom.
>
> If code cannot be confidently understood, it will be accidentally broken.
>
> Therefore, *Arrange the important parts of the code so it fits on one page. Make that code understandable to a person reading it from top to bottom. Do not require the code to be repeatedly scanned in order to understand how data is used and how control moves about.*
>
> This pattern can be achieved by using the following patterns: Local Variables Defined and Used on One Page, which tries to keep local variables on one page; Assign Variables Once, which tries to minimize code scanning by having variables changed just once; Local Variables Reassigned Above their Uses, which tries to make a variable's value apparent before its value is used while scanning from top to bottom; Make Loops Apparent, which helps people understand parts of a program that are non-linear while retaining the ability to scan them linearly; and Use Functions for Loops, which packages complex loop structure involving several state variables into chunks, each of which can be easily understood. (Gabriel, 1995)

The humanity of this pattern shines through in the words *confidently*, *stress*, *secure*, *culture*, *understand*, etc. Human comfort emerges from the aesthetics of the design.

Aesthetics isn't just about code formatting, either. Aesthetics might be founded in sound structure, intuitive module interfaces, or a holistic concern for the system and its environment. We find the highest principles of architecture in the classic Vitruvian triad: utility, firmness, and last but not least, delight—or aesthetics.

There is an even more obvious side to aesthetics: how users view our programs. An aesthetically pleasing human interface makes life more enjoyable for the users of the program: they become less frustrated, more productive, and

perhaps even more entertained than they would otherwise be.

Consider the following three patterns, abstracted from Ward Cunningham's CHECKS pattern language. These patterns ostensibly deal with human interfaces and the aesthetics of human-computer interaction:

> **1. Whole Value**
>
> **. . .**
>
> When parameterizing or otherwise quantifying a business (domain) model, there remains an overwhelming desire to express these parameters [currency, calendar, periods, telephone numbers] in the most fundamental units of computation. Not only is this no longer necessary (it was standard practice in languages with weak or no abstraction), it actually interferes with smooth and proper communication between the parts of your program and between the program and its users. Because bits, strings, and numbers can be used to represent almost anything, any one in isolation means almost nothing.
>
> Therefore: Construct specialized values to quantify your domain model and use these values as the arguments of their messages and as the units of input and output.…
>
> **2. Exceptional Value**
>
> A business model will normally be composed of a basic case or abstraction that is specialized and/or refined to capture the diversity present in the business. However, there will often be circumstances where the inclusion of all business possibilities in the class hierarchy would be confusing, difficult, or otherwise inappropriate. You will therefore at times need to extend the range of an attribute beyond that offered by a Whole Value (1). Consider a pollster who collects answers like agree, strongly agree," and so on. Answers that defy quantification, like "illegible" or "refused," are better represented outside the range of values, no matter how fuzzy they may be. However, the structure of a domain model should save a place for this sort of missing data, for it may appear later. In fact, missing values are impossible to avoid during the creation (data entry) of all but the most trivial domain models.
>
> Therefore: use one or more distinguished values to represent exceptional circumstances. Exceptional values should either accept all messages, answering most of them with another exceptional value, or reject all messages...
>
> **3. Meaningless Behavior**
>
> Given that Whole Values (1) used to quantify your business logic will exhibit subtle variations in behavior and that Exceptional Values (2) may appear throughout the computations, it is possible that the methods you write will stumble in circumstances you cannot foresee. Keep in mind that the rules of business apply only selectively, and that the evolution of your business practices can wiggle around even those rules that "must" apply. In your domain models you are chartered to express business logic with no more complexity than its original conception or current expression.
>
> **Therefore**: Write methods without concern for possible failure. Expect the input/output widgets that initiate computation to recover from failure and continue processing. Output will remain blank, because any other output would be an attempt to attach meaning to meaningless behavior. Users will interpret unexpected blanks to mean that inputs do not apply and/or outputs are unavailable... (Cunningham, 1995)

The user benefits from a clean, intuitive interface. Some interfaces suffer from code that tries to help the user too much by popping up error windows every time the application trips over a mistake; this pattern avoids that.

Although ostensibly about human interface design, these patterns benefit the designer as much as the end user. The pattern says don't worry about error handling: propagate problems to the end of the evaluation chain (the human interface) and deal with the problems just before they reach the user. The patterns tell both how to propagate the errors and how to catch them at just the right moment. This relieves programmers of the cognitive burden of error management as they write code and leaves the code easier to maintain afterward. Lacking scattered conditionals that check for error conditions, the code is much more concise and readable than if brute-force techniques were used.

Most software patterns are about software architecture, and patterns certainly take their inspiration from the more ancient field of building architecture. Aesthetics are key to good architecture. Rybczynski noted that architecture is the sum of engineering and *culture*; we rarely admit the latter in our design methods. He illustrates this principle with a powerful example:

> The communication of meaning, more than beauty, distinguishes architecture from engineering. A bridge must be solid, functional, and attractive; a good public library must be all of these, but it also carries cultural baggage. Its architecture defines our attitude toward reading and celebrates a sense of civic pride. A library is more than a warehouse for books; it is a built evocation of an intellectual ideal. (Rybczynski, 1992: pp. 266—267)

We are here to give our software *meaning* in the societies in serves, not just to build the equivalent of nuts and bolts. Meaningful systems serve their users well. During the construction of a meaningful system, the system builder and system user are engaged in a (sometimes implicit) dialogue where each must understand the patterns of the other. Software authors must understand the user domain and vocabulary; the software user has to understand the meaning of mouse movements and key clicks as the author has mapped them into the user vocabulary. Patterns can capture the subtle aspects of this dialogue at a high level, aiding this communication between user and author. Such concerns distinguish merely functional systems from systems that are pleasant to use.

7.9. Interdisciplinary Scope

The software pattern community took its foundations from a foreign field: building architecture and urban planning. Software patterns are broadening into the domains of organization and process, training, and areas outside the traditional focus of software discipline. But we aspire to go even further.

There is a traditional and frequently highlighted gap between the domain of software designers and the domains of their customers. Software designers traditionally know data structures, algorithms, paradigms, and methods; domain practitioners understand the practices of their business, craft, or pastime. Software developers will better be able to meet customer needs by understanding their domain needs. Going even further, if we can capture the important software design patterns that we have found useful for solutions in a specific domain like biology, we can give them those patterns as they start to do their own programming. This is consistent with the architect who employs the patterns of the predominant architectures of a culture, so people can participate materially in the construction of their own homes, churches, factories, and parks. Patterns provide a vocabulary for customer engagement, but we must first engage our customers in biology, telecommunications, finances, aerospace, and other disciplines to work with them to develop this vocabulary.

Customer engagement is a popular agenda in contemporary management circles. We can build on other parallels that are less direct, but perhaps even more powerful, by looking for parallel design structures that transcend domains.

As an example, consider the CHECKS pattern language (see page 41), which applies to the design of software that works with interactive user interfaces. This pattern is fundamentally the same pattern as the IEEE floating point standard, with obvious parallels between "not a number" and "exceptional value."

We find parallels for Leaky Bucket Counter (page 26) in other domains as well. Leaky Bucket Counters do the same thing in software that low-pass filters do in analogue circuits and stereo systems. These related patterns sometimes point to broader patterns that transcend domains; this is a fundamental organizing principle for pattern systems (Classifying Patterns on page 23).

The most powerful patterns touch human and technological concerns at once. Short of attacking two domains with one pattern, we can carefully weave patterns from multiple domains into a cogent language that integrates human concerns with technology. Steve Berczuk has done this by integrating his earlier design patterns with organizational and process patterns (Berczuk, 1996b):

> Steve "includes social context as a motivating context for a pattern." He observes that both organizational problems and technical problems figure heavily in satellite telemetry software development, largely because such projects employ geographically distributed development teams. These projects lack a single architectural authority, making it difficult to apply the pattern *Architect Controls Product*. Because groups are geographically isolated and have widely divergent interests, it's important to localize changes to the software owned by a team *(Code Ownership, Organization Follows Location*). The solution is to decouple teams through flexible architectural interfaces, employing patterns like *Callback, Parser Builder, and Hierarchy of Factories*. The resulting architecture and organization help reinforce patterns *like Developer Controls Process* and *Code Ownership*. (Coplien, 1995b)

7.10. Ethics

Software quality is subjective, and software intellectual property is an ongoing challenge for the legal profession. The issue has gained notoriety through look-and-feel suits, through the Free Software Foundation, the League for Programming Freedom, and other groups. Comparable techniques, claims, methods and tools defy objective measures, yet they vary dramatically in quality and suitability.

This is a market ripe for disaster, a market where unproven techniques, methods and tools can gain large foot-holds. Patterns are particularly subject to the same abuses, particularly because they are so subjective, and because they have such allure both to informed designers and to the uninitiated.

For these reasons, Norm Kerth led an ethics session an the first conference on Pattern Languages of Programming (PLoP) in 1994, so patterns have had a strong ethical foundation since their earliest years in software. This section looks at the ethical aspects of the pattern value system.

7.10.1. Intellectual Currency

Many patterns tackle problems whose solutions are difficult to measure directly. How does one measure the quality of a design? In the long run, one can measure how well it satisfies a customer and how well its structure stands up under evolution (return on investment, net present value, etc.). But such measures are trailing indicators, not the stuff patterns are made of. Patterns capture the insights of experts with good track records. Although patterns are supported by a rationale, the decision to use one design technique over another ultimately comes down to a subjective evaluation. Yet these intellectual gems are sometimes the key design insights that enable a product or a technology, and we should honor those who take the time to codify these techniques as patterns.

Mary Shaw uses the term intellectual currency to describe a model for disseminating the literature of technical innovation. Normal currency leaves you poorer if you give it away. Attributed innovative ideas are worth more to the originator when given away, that is, fairly, generously, and accurately acknowledged. Pattern writers and pattern users are well advised to cite their sources when using or codifying patterns. It encourages our experts to share their secrets with us.

7.10.2. Legal Issues

Many people ask how copyrights and patents relate to patterns. Patterns can be copyrighted as can any other piece of literature. Patterns may describe technical solutions worthy of patentability independent of their expression in pattern form.

Patterns are not inventions in themselves; rather, they codify well-established (although often obscure) practice. Someone might be identified as the first to think of a pattern; another as the first to use it; another as the first to discover that it is a recurring pattern; another as the first to write it down in pattern form; another as the first to publish it; others as the first publishers of derivative works; etc. Each of these roles may play a part in intellectual property rights.

Legal issues are driven largely by precedent and convention as well as by law, and a legal professional should be consulted for binding legal advice in the appropriate jurisdictions.

7.11.3. Fighting Information Hiding

Knowledge is power. Individuals with key technical knowledge too often look to their expertise as a source of their own job security and perhaps more deeply as a source of personal security and identity.

As we gather patterns from industry experts, we must make them feel secure enough to share their knowledge with others: this is the primary motivation for the principle of intellectual currency.

Beyond attribution, we must persuade experts that they can be even more effective if they share their ideas broadly than if they are the sole interpreters of their expertise. Written knowledge can outlast these experts chronologically and can outpace them geographically.

Last, even patterns can't replace experts. We capture patterns from the experts to help work-a-day engineers amplify what they already know, and to draw out the skills within them. A true expert uses patterns unconsciously, as they have become part of the fabric of the expert's mental processes. Alexander calls this passing through "the gate" (Alexander, 1979: pp. 546—549). Patterns capture edges of the intuitive problem-solving strategies of true experts; much of the power lies an a deeper level, accessible only to experts reaching within themselves to draw on bits inaccessible to the rest of us. Patterns can help the work-a-day engineer avoid inexpert errors, but they won't transform every carpenter into a Frank Lloyd Wright or every software engineer into a Donald Edmund Knuth. We still depend desperately on the cultivated expertise of domain experts.

7.10.4. Don't Hype

Hype was an important topic of discussion an Norm Kerth's ethics session. Hype has two sources: enthusiasm that stems from well-informed advocates of patterns and inflated expectations from practitioners outside the pattern community, whether well-intentioned or not. Hype often propels expectations beyond what patterns can deliver.

By making hype unfashionable in the pattern community, we hope to minimize both sources of hype, and to increase the chances that expectations will match what patterns can deliver. It's inevitable that opportunists will raise expectations about patterns to sell their wares, and in fact, this problem may be beyond the influence of the pattern community.

There is another twist on the hype issue: undue attention to reducing hype is a form of hype in itself.

7.10.5. Aggressive Disregard for Originality

Unproven software ideas gain ground too easily. In the long term, some ideas gain a reputation as being unworkable or inferior, which causes a credibility gap for the original proponent. It seems like many software ideals fall into this trap. We seek to avoid this trap in patterns by building on proven ideas, rather than by pioneering new ideas.

This value is explored further in Section 7.6.

8. History

Interest in software patterns emerged from the leading software designers in the industry. Because objects have been the focus of leading software design and practice since the mid 1980s, it was natural for patterns to emerge from object-oriented design. The link between patterns and objects persists to this day, although there is nothing intrinsic about the relationship.

Some of the earliest work with patterns was done by Ward Cunningham and Kent Beck, who were then at Tektronix. Kent had encountered patterns in his early academic days; he writes:

I first discovered patterns as an undergraduate at the University of Oregon. Many of the students in my freshman dorm ... were in the School of Architecture. Since I had been drawing goofy house plans since I was six or seven, they pointed me in the direction of Christopher Alexander. I read all of The Timeless Way of Building standing up in the university bookstore over the course of several months.

I had been working at Tektronix for a year and a half when I came across Alexander again. I found a battered old copy of *Notes on the Synthesis of Form* in Powell's. Alexander's excoriation of methodologists in the introduction to the second edition resonated with my biases, leading me to Timeless Way again. It seemed everything he didn't like about architects, I didn't like about software engineers. I convinced Ward Cunningham that we were onto something big. (Portland pattern repository, http://c2.com/ppr)

In 1987, Ward Cunningham and Kent Beck were consulting with a group that was having trouble designing a user interface They decided, in Ward's VW Vanagon on the way over, to try out the pattern stuff they'd been studying. Alexander said the occupiers of a building should design it, so they had the users of the system design the interface. Ward came up with a five-pattern language that helped them take advantage of Smalltalk's strengths and avoid its weaknesses. The patterns were originally presented verbally. Here are those patterns, extracted from the Wiki-WikiWeb web site (http://c2.com/cgi/wiki?WindowPerTask):

> **Window Per Task**:
>
> Make a specific window for each task the user must perform. All of the information needed to complete a task should be available in the Few Panes of the window. Assume prerequisite tasks have been completed (if they haven't the user will simply change windows).
>
> This pattern effectively side-steps gross problems that Model-View-Controller had with mutually dependent window at the time of the Tek LT1000 project.
>
> The LT1000 domain engineers knew exactly what tasks their users performed. There were only five or six.
>
> **Few Panes**:
>
> To understand complex things one often must see it from several points of view. **Therefore**: Provide these points of view (called simply View) by dividing the area of your Window Per Task into panes.
>
> **Standard Panes**:

One must learn to operate each kind of pane offered in the Few Panes of every window.

Therefore: Cast each pane into the format offered by one of a few standard panes. For Test Programming, these should be limited to:

- Text
- List
- Table
- Waveform

Nouns And Verbs:

Things exist while action happens.

Therefore: Put lists of things (nouns) in a list pane (one of Few Panes) which persists through interactions. Put actions (verbs) in Short Menus which pop up and then disappear as the action commences.

The one time our domain specialists had trouble satisfying this pattern was for a menu they thought should include:

- Decimal
- Octal
- Hex

We recognized that they were in fact thinking of the menu options as actions and suggested that they rename them:

- Be Decimal
- Be Octal
- Be Hex

Short Menus:

The elements of a pop-up menu must be visually searched repeatedly.

Therefore Make them short, fixed and single-level.

It's interesting that this pattern was easily met because of the conditions set up by Window Per Task and Nouns and Verbs. [From the WikiWikiWeb, http://c2.com/cgi/wiki]

They were amazed at the (admittedly spartan) elegance of the interface their users designed. They reported the results of this experiment at OOPSLA '87 in Orlando. Ward wrote a panel position (Cunningham, 1988), and both Kent and Ward presented at Norm Kerth's workshop "Where do objects come from?" (Beck, 1988). They talked patterns until they were blue in the face, but without more concrete patterns, nobody was signing up.

Meanwhile Erich Gamma was busy writing and reflecting about object-oriented design in ET++ as part of his Ph.D. thesis. ET++ had already established itself as an exemplary framework in the C++ community. Erich had realized the importance of the recurring design structures, or patterns, of this framework. The question really was: how do you capture and communicate them?

In 1990 at ECOOP/OOPSLA in Ottawa, Bruce Anderson ran a birds-of-a-feather session called "Toward an Architecture Handbook" where he, Erich Gamma, Richard Helm, and others got into discussions about patterns.

Just prior to ECOOP'91 Erich Gamma and Richard Helm, sitting on a rooftop in Zurich on a sweltering summer's day, put together the very humble beginnings of the catalog of patterns that would eventually become Design Patterns. There they identified many patterns including such familiar and unfamiliar patterns as:

1. Composite
2. Decider
3. Observer
4. Constrainer

Many of these patterns made ii into Design Patterns (Gamma et al., 1995); many others remain rough and unpublished to this day.

Things really got rolling at the OOPSLA workshop that Anderson ran in 1991. Coincidentally, Erich Gamma,

Richard Helm, Ralph Johnson, and John Vlissides were all there; they would later become the “Gang of Four” that wrote Design Patterns. Most of the Hillsiders-to-be were there: Ward and Kent, Desmond D’Souza, Norm Kerth, and other pattern notables like Doug Lea and Wolfgang Pree. Bruce repeated the workshop in 1992, which is where the Gang of Four properly got together. Frank Buschmann’s first publication on patterns was presented there as well.

In late 1988, the writer started cataloging language-specific C++ patterns he called idioms, which are the lowest-level patterns. Early manuscripts of this work were used to teach objects and C++ an AT&T in early 1989. Addison-Wesley published the book in September 1991 *as Advanced C++ Programming Styles and Idioms*. Peter Coad had been exploring patterns in parallel as well. He mentioned them in a 1991 issue of his newsletter, and published an article in *Communications of the ACM* in 1992 (Coad, 1992).

By this time, pattern folks started discovering their mutual interest and sought opportunities to take their ideas a step further. In May of 1993, some folks got together for a workshop on object-oriented design an IBM in Thornwood, New York. Reflective thinking was a big part of this workshop, both for the standard curriculum and for the extracurricular pattern discussions after hours. Desmond D’Souza, Doug Lea, Kent Beck, Ralph Johnson, Bruce Anderson, Ron Casselman, and John Vlissides were the facilitators.

In August that same year, Kent and Grady Booch sponsored a mountain retreat in Colorado where a group of us converged on foundations for software patterns. Ward Cunningham. Ralph Johnson, Ken Auer, Hal Hildebrand, Grady Booch, Kent Beck and the writer struggled with Alexander’s ideas and our own experiences to forge a marriage of objects and patterns. We agreed that we were ready to build on Erich Gamma’s foundation work studying object-oriented patterns to use patterns in a generative way in the sense that Christopher Alexander uses patterns for urban planning and building architecture. We then used the term generative to mean “creational” to distinguish them from “Gamma patterns” that captured observations.

To better understand Alexander’s patterns, the group conducted an exercise on a hillside to design a building called the Center for Object-Oriented Programming. Our vision was to design a building where software practitioners could come together with their clients to learn each others’ worlds better, using patterns as the basis for the dialogue. The group designed a building as Alexander suggests it should be done: laying out the building on the actual construction site, taking advantage of the landscape, blending the building with its surroundings. It was an enlightening shared experience. This group would continue to meet, hone their understanding of patterns, and carry patterns forward into the industry. We started informally referring to ourselves as The Hillside Group.

Bruce again held his workshop an OOPSLA ‘93, this time with patterns in the workshop title and prominently on the agenda.

The Hillside Group men again in early April 1994 to “plan” the first Pattern Languages of Programming (PLoP) conference. We wanted something really wacky and unusual, but most of us felt (and were willing to take) the risk that goes with new things. That was Richard Gabriel’s first time with us. He exhorted us all to go into PLoP with confidence and act as though we knew what we were doing.

On August 4, about 80 people came together at the Allerton Park estate near Monticello, Illinois, to do just that. Things went well—the weather even cooperated. Ward Cunningham and Ralph Johnson were program and conference chair, respectively. Kent Beck, who had just welcomed a new addition to his family, couldn’t make it, reminiscent of OOPSLA in New Orleans in 1989. The PLoP proceedings came out in May, 1995, *as*

Pattern Languages of Program Design. (Coplien & Schmidt, 1995)

In the meantime, the Gang of Four had wrapped up their work and sent in to the publisher. The first major compendium of patterns between two covers, *Design Patterns: Elements of Reusable Object-Oriented Software* (Gamma et al., 1995) made it out in time for OOPSLA '94. It sold 750 copies an the conference, more than seven times the highest number of any technical book Addison-Wesley had ever sold an a conference. The book is still reported to be doing well.

References

Adams, M., J. Coplien, R. Gamoke, R. Hanmer, F. Keeve, and K. Nicodemus. (1996). Fault-Tolerant Telecommunication Patterns. Pattern Languages of Program Design—2, Reading, MA: Addison Wesley.

Alexander, C. (1974). Notes on the Synthesis of Form. Cambridge, MA: Harvard University Press.

Alexander, C. et al. (1977). A Pattern Language. New York: Oxford University Press.

Alexander, C. (1979). The Timeless Way of Building. New York: Oxford University Press.

Anthony, D. L. G. Patterns in Classroom Education. Pattern Languages of Program Design—2, Reading, MA: Addison Wesley.

Beck, K.. R Crocker, J. Coplien, L. Dominick, G. Meszaros, E Paulisch, and J. Vlissides. (1996). Industrial Experience with Design Patterns. Proceedings of ICSE '96.

Beck, K. (1988). Using a pattern language for programming. In Workshop on Specification and Design, organized by Norman Kerth, A CM SIGPLAN Notices 23,5 (Addendum to the Proceedings of OOPSLA '87).

Berczuk, S. (1996). Book Review: 'Object Models: Strategies, Patterns and Applications." Object-Oriented systems 43).

Berczuk, S. (1996). Organizational Multiplexing: Patterns for Processing Satellite Telemetry with Distributed Teams. Pattern Languages of Program Design—2, Reading, MA: Addison Wesley.

Booch, G. (1994). Object-Oriented Analysis and Design with Applications. Redwood City, CA: Benjamin Cummings, 2nd ed.

Buschmann, F. and R. Meunier. (1995). A System of Patterns, Pattern Languages of Program Design, Reading, MA: Addison Wesley.

Coad, P., D. North, and M. Mayfield. (1995). Object Models: Strategies, Patterns and Applications. Englewood Cliffs, NJ: Prentice Hall.

Coad, P. (1992). Object-oriented patterns. Communications of the ACM35(9).

Coplien, J. 0. (1992). Advanced C++ Programming Styles and Idioms. Reading, MA: Addison-Wesley.

Coplien, J. 0. (1995). A Development Process Generative Pattern Language. Pattern Languages of Program Design, Reading, MA: Addison Wesley.

Coplien, J. 0. (1996). The Human Side of Patterns. C++ Report, January, 81—86.

Cunningham, W (1988). Panel on Design Methodology. ACM SIGPLAN Notices 23(5) (Addendum to the Proceedings of OOPSLX87).

Cunningham, W. (1995). The CHECKS Pattern Language of Information Integrity. Pattern Languages of Program Design, Reading, MA: Addison Wesley.

DeBruler, D. L. (1995). A Generative Pattern Language for Distributed Processing. Pattern Languages of Program Design, Reading, MA: Addison Wesley.

Edwards, S. H. (1992). Streams: A Pattern for "Pull-Driven" Processing. Pattern Languages of Program Design, Reading, MA: Addison Wesley.

Gabriel, R. P. (1996). Patterns of Software: Tales from the Software Community New York: Oxford University Press.

Gamma, B., R. Helm, R. Johnson, and J. Vlissides. (1995). Design Patterns: Elements of Reusable Object-Oriented Software. Reading, MA: Addison-Wesley.

Jacobson, I., M. Chisterson, P. Jonsson, and G. Overgaard, (1992). Object-Oriented Software Engineering—A Use Case Driven Approach. Wokingham, England: Addison-Wesley.

Kerth, N. (1995). Caterpillar's Fate. Pattern Languages of Program Design, Reading, MA: Addison Wesley.

Knuth, D. B. (1991). Literate Programming. Stanford, CA: Center for the Study of Language and Information.

Koenig, A. R. (1995). Patterns and Antipatterns. Journal of Object-Oriented Programming 8(1).

Meszaros, G. (1996). A Pattern Language for Improving Capacity of Real-time Systems. Pattern Languages of Program Design—2, Reading, MA: Addison Wesley.

Morgan, M., translator. (1960). Vitruvius: The Ten Books of Architecture. New York: Dover.

Parnas, D. L. (1976). On the Design and Development of Program Families. IEEE Transactions on Software Engineering, SE-2:1—9.

Pree, W (1995). Design Patterns for Object-Oriented Software Development. Wokingham, England: Addison-Wesley.

Rumbaugh, J., M. Blaha, W. Premerlani, F. Eddy, and W Lorenson. (1991). Object-Oriented Modeling and Design. Prentice Hall, Englewood Cliffs, NJ.

Rybczynski, W (1989). The Most Beautiful House in the World. New York: Viking.

Rybczynski, W (1992). Looking Around. New York: Viking.

Sengé, P. (1990). The Fifth Discipline: The Art and Practice of the Learning Organization. New York: Doubleday.

Viljamaa, P. (1995). The Patterns Business: Impressions from PLoP-94. ACM Software Engineering Notes 20(1).

Vlissides, J., N. Kerth and J. Coplien, Eds. (1996). Pattern Languages of Program Design—2. Reading, MA: Addison-Wesley.

Volk, T (1995). Metapatterns across Space, Time and Mind New York: Columbia University Press.

Wolf, K. and C. Liu. (1995). New Clients with Old Servers: A Pattern Language for Client/Server Frame-

works. Pattern Languages of Program Design, Reading, MA: Addison Wesley.

Weinberg, G. M., and D. (1988). Weinberg. General Principles of System Design. New York: Dorset.

Zimmer, W. (1995). Relationships Between Design Patterns. Pattern Languages of Program Design, Reading, MA: Addison Wesley.

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

0

P

Q

R

S

T

U

V

W

Z